AGH University of Science and Technology

Faculty of Electrical Engineering, Automatics,
Computer Science and Electronics

Sławomir Zieliński

Architecture of a Framework for Dynamic, Semantics Based Deployment of Overlay Networks in Peer-to-Peer Environments

Doctoral Thesis

Supervisor: prof. dr hab. inż. Krzysztof Zieliński

Kraków, 2009

Acknowledgements

First of all, I would like to thank my supervisor, prof. Krzysztof Zieliński, who has helped me with his inspiration, constructive criticism and guidance. His insight was of invaluable help for me to complete my research presented in this thesis.

Thanks to all my colleagues – former and current members of the Distributed Systems Research Group at the Computer Science Department of AGH University of Science and Technology – for their support of my work.

Finally, I would like to thank my family for their understanding, endless patience and encouragement that upheld me during the time I spent on working on this thesis.

Contents

1. Introduction 9
1.1. Key definitions 11
1.2. Overview of framework goals 12
1.3. Outline of supported message processing systems functionality 14
1.3.1. Selection of message sources and filtration of messages 14
1.3.2. Message processing versus message passing 16
1.3.3. Component deployment 16
1.3.4. Inter-component communication 17
1.4. The thesis and research goals 18
1.5. Original achievements 18
1.6. Organization of the thesis 19
2. Survey of related middleware environments 20
2.1. Key concepts of existing message passing and processing systems 20
2.1.1. Message stream processing models 20
2.1.2. Message routing 24
2.1.3. Message processing and reuse of intermediary results 28
2.1.4. Quality of service 29
2.1.5. Choices for the proposed framework 30
2.2. Metadata representation and distribution 32
2.2.1. Externalization of type systems 33
2.2.2. Message structure 33
2.2.3. Representation of metadata related to document syntax 34
2.2.4. Representation of metadata related to document semantics 36
2.2.5. Choices for the proposed framework 37
2.3. Runtime environment virtualization 38
2.3.1. Overlay networks 38
2.3.2. Addressing and communication patterns 40
2.3.3. Choices for the message processing overlay networks 44
2.4. Adaptability of framework instances 45
2.4.1. Introspection 45
2.4.2. Adaptation 46
2.4.3. Reflection 46
2.4.4. Security 47
2.4.5. Conclusions for the proposed framework 48
2.5. Summary 48
3. Framework instance model 50
3.1. Overview of the framework instance architecture 50
3.2. Overlay network creation 53

3.2.1. Use cases.....53
3.2.2. Operation of key internal services.....59
3.3. Overlay network maintenance.....66
3.4. Framework instance maintenance.....69
3.5. Framework instance extensibility.....75
3.5.1. Ease of integration with existing systems.....75
3.5.2. Framework customization.....75
3.6. Summary.....77
4. Prototype implementation 78
4.1. Technologies used for prototype implementation.....78
4.1.1. JXTA.....78
4.1.2. XMLBeans toolkit.....82
4.1.3. IODT Minerva.....82
4.2. A minimal configuration of framework prototype instance.....82
4.3. Implementation of key use cases.....83
4.3.1. Use cases: "produce messages", "get available message streams".....83
4.3.2. Use case: "create a message processing overlay network".....86
4.4. Key internal services.....100
4.5. Summary.....104
5. Evaluation of framework prototype 105
5.1. Marathon reporting service case study.....105
5.1.1. Marathon reporting service model.....106
5.1.2. Implementation of the service based on the REMP prototype.....109
5.1.3. Evaluation environment and overlay network startup.....117
5.1.4. Evaluation scenarios.....120
5.1.5. Evaluation results.....124
5.2. Examples of other applications.....135
5.2.1. Stock monitoring system.....135
5.2.2. Accident rescue system.....137
5.3. Summary.....139
6. Conclusions 140
6.1. Research achievements.....140
6.2. Framework shortcomings and future work.....142
References 144

List of Figures

Fig. 1. The main actors of a system built upon the proposed framework. 12
Fig. 2. The idea of an overlay network implementing the functionality of a virtual message producer. 13
Fig. 3. Message processing system architecture overview. 13
Fig. 4. Message flow in the CORBA Notification Service. 21
Fig. 5. An example of Gryphon's information flow graph. 22
Fig. 6. Waste of resources in a pure flooding-based approach (example). 25
Fig. 7. Filtration-based routing (example). 26
Fig. 8. Covering-based subscription routing (example). 27
Fig. 9. Subscription forwarding based on source advertisements (example). 28
Fig. 10. The structure of CORBA Notification Service message. 34
Fig. 11. The idea of middleware-implemented overlay networks. 39
Fig. 12. Mapping between a logical communication channel and multiple transport connections. 39
Fig. 13. The subsystems building up a REMP framework instance. 50
Fig. 14. The life cycle of an overlay network created by framework instances. 51
Fig. 15. Key elements of a running message processing system infrastructure. 52
Fig. 16. Services and API elements provided by the framework. 53
Fig. 17. The basic use cases for the system. 54
Fig. 18. A simplified collaboration diagram for the "Produce messages" use case. 55
Fig. 19. The process of message processing overlay network creation (simplified). 56
Fig. 20. Consumer-initiated introspection regarding contract offers (simplified). 59
Fig. 21. Matching service operation (simplified). 60
Fig. 22. Generation of a message processing node's code. 61
Fig. 23. The process of storing generated component code (simplified). 62
Fig. 24. Deployment and instantiation of a message processing node. 63
Fig. 25. A basic framework instance container. 63
Fig. 26. Creation of a communication channel. 64
Fig. 27. A simple use case for the monitoring and metering service. 67
Fig. 28. A simplified node migration process. 68
Fig. 29. Operation of checkpointing service (simplified). 71
Fig. 30. Worker selection in services based on the proxy-worker pattern. 72
Fig. 31. Cooperation between redundancy service enabled instances and user message flow. 74
Fig. 32. Transparent load balancing using redundancy service. 74
Fig. 33. Overview of framework's prototype implementation architecture. 78
Fig. 34. Advertisement of an existing overlay network. 81
Fig. 35. Prototype matching service worker capable of executing concept-based queries. 82

Fig. 36. A minimum set of peers constituting internals of a fully functional framework instance....83
Fig. 37. Message producer application programmer interface....84
Fig. 38. Sequence diagram for interaction between a message producer and Publish Service. 84
Fig. 39. Contents of message source capabilities description....85
Fig. 40. Message source advertisement publication and retrieval (example). 86
Fig. 41. Message consumer application programmer interface. 86
Fig. 42. Sequence diagram for interaction between a message consumer and Subscribe Service....87
Fig. 43. Contents of the overlay network specification....88
Fig. 44. Resolution of a sub-query regarding raw message sources. 89
Fig. 45. Matching service worker's programmer interface. 90
Fig. 46. Generation of a deployable component based on processing component specification provided by user (simplified). 91
Fig. 47. Contents of the processing component specification. 91
Fig. 48. Contents of the input and output port specifications. 92
Fig. 49. Wrappers created by the framework's prototype component generation service worker. 93
Fig. 50. Module storage in reference code repository service worker. 95
Fig. 51. Instantiation of a new module (simplified)....96
Fig. 52. Registration of component wrappers in container instances of publish and subscribe services....97
Fig. 53. Options for selecting sending ends of the inbound communication channels. 98
Fig. 54. Creation of a single message channel (simplified)....99
Fig. 55. State diagram for an SRP-enabled node. 101
Fig. 56. Generic sequence diagram for obtaining remote module reports. 103
Fig. 57. Components that build up the journalist's application. 107
Fig. 58. Components that build up the professional runner's application....107
Fig. 59. Components that build up the club manager's application. 108
Fig. 60. Components that build up the contender fan's application. 108
Fig. 61. Marathon reporting service overlay network....109
Fig. 62. The ontologies published by message sources....111
Fig. 63. Evaluation environment network topology. 117
Fig. 64. Configuration of Milano City Marathon simulation case study – raw message sources....122
Fig. 65. The number of active (publishing) raw message sources during the marathon results reporting simulation. 123
Fig. 66. The expected numbers of messages to be published by raw message sources....123
Fig. 67. Average numbers of messages to be published by active message sources. 124
Fig. 68. Directly connected consumer heap size (experiment 4). 125

Fig. 69. Professional runner consumer's user interface heap size (experiment 11). 125
Fig. 70. Deployment service worker heap size (experiment 14, "top 10" filter and average pace counter deployed). 126
Fig. 71. The numbers of messages received by a consumer in a "direct 1:1" scenario (experiment 2). 127
Fig. 72. The numbers of messages received by a consumer connected through a lower speed link in a "direct 1:5" scenario (experiment 4). 127
Fig. 73. The numbers of messages received by a consumer connected through a restricted link in a "direct 1:10" experiment (communication channel from message source "43" failed to set up). 128
Fig. 74. The numbers of messages received by one of professional runner consumers connected through the 640kbps link. 128
Fig. 75. Numbers of messages received by a professional runner client application connected through the 640 kbps link. 129
Fig. 76. Producer application CPU usage (experiment 3 - "direct 1:1" scenario). 130
Fig. 77. Producer application CPU usage (experiment 5 - "direct 1:5" scenario). 130
Fig. 78. Producer application CPU usage (experiment 6 - "direct 1:10" scenario). 131
Fig. 79. Consumer application CPU usage (experiment 6 – "direct 1:10" scenario). 131
Fig. 80. Producer application CPU usage (experiment 8 - two processing components attached). 132
Fig. 81. Deployment service worker CPU usage (experiment 9 - one processing components deployed, 1 receiver attached). 132
Fig. 82. Deployment service worker CPU usage (experiment 15 – one processing component deployed, 10 receivers attached). 133
Fig. 83. Second-stage deployment service worker CPU usage (experiment 9 - 1 nationality filter deployed). 133
Fig. 84. The effect of delegation of message processing on message consumer's memory consumption. 134
Fig. 85. The effect of delegation of message processing on message producer's CPU utilization. 134
Fig. 86. The effect of delegation of message processing on the number of messages received by end user's application. 135
Fig. 87. The hypothetical flow of accident reporting messages. 138

List of Tables

Table 1. Types of addresses to be used by framework instances. 45
Table 2. The main concepts of the reviewed technologies to be reused by REMP framework. 48
Table 3. Hardware and software configurations of the hosts present in evaluation environment. 118
Table 4. Configuration of the REMP instance used for evaluation. 119
Table 5. The experiments conucted in the evaluation phase. 120

List of Listings

Listing 1. Operations performed by the overlay network manager when creating a new overlay network. 58
Listing 2. Java interface to be implemented by processing nodes compatible with the framework prototype. 94
Listing 3. Prototype component wrapper interface. 94
Listing 4. Extended JXTA peer group interface. 97
Listing 5. A sample configuration file for the message producer application. 110
Listing 6. The elements of a single contender result message. 111
Listing 7. Excerpts of the “top 10” filter code. 113
Listing 8. The code that fills up one inbound port specification (link criteria omitted). 114
Listing 9. Creation of a message source specification. 115
Listing 10. Specification of the arguments needed to create an internal link (link criteria omitted). 116
Listing 11. A part of default configuration file for REMP elements. 118
Listing 12. A simple set of rules for automated stock market application. 136

1. Introduction

Peer-to-peer file sharing networks caught on not because the resources offered by them could not be obtained in a different way. One of the main reasons of their popularity was their ease of use. A user of a typical file-sharing network does not need to know the details of file and file transfer handling. In order to use the service, the user needs only to install a piece of software and follow a few simple configuration steps. The simplicity of the service interface encouraged creation of communities of common interest, targeted at file sharing. The work presented in this thesis was partially motivated by that phenomenon.

Distributed systems benefit from collaboration between remote nodes. Depending on the scenario, the nodes can be specialized in providing particular functionality (e.g. storage nodes) or just provide others with their spare computing power (as in the case of SETI@home[SETI] project workers). In order to organize the collaboration, virtually all distributed systems rely heavily on underlying communication facilities. The facilities are responsible for solving the crucial problem of passing pieces of information from one system entity to another in context of particular application. The application context defines not only the syntax of messages, but also their semantics and sequences, i.e. protocols that are used between entities.

This thesis introduces REMP – REmote Message Processing framework. REMP allows for delegation of message processing components to remote devices, which is a set of middleware services based on peer-to-peer networks. Its goal is to create an easy-to-use, but flexible peer-to-peer framework for programmers and in this way encourage creation of communities of common interest targeted at execution of distributed applications. The framework overcomes a few common disadvantages that are present in contemporary peer-to-peer platforms:

- insufficient expressiveness of commonly used meta-information schemas,
- insufficient expressiveness of commonly used message filter definition languages,
- insufficient support for creating Internet-wide generic application execution environments,
- insufficient opacity of middleware environments.

Meta-information. The meta-information regarding the syntax and semantics of messages can be either hidden in the processing code, or externalized. Externalization of meta-data provides for integration and eases system evolution. Typically the meta-data concerns message schemas or type systems. In order to facilitate system evolution better, the externalized data should be annotated semantically. A classic approach is to provide human-readable documents which explain the meaning of data types, object fields, etc. Although useful, such approach leaves almost no possibility for machine processing of the meta-data. Therefore, automatic reasoning upon the elements of the distributed system is hardly possible. The work presented in this thesis shows that by using semantic metadata to describe message streams helps the distributed system developer to select appropriate sources of messages.

Filtering and processing of messages. Application context is another factor that frequently decides upon relevance of messages. For example, if a particular application is interested in receiving maximum value of some variable, the current maximum could be used as a criteria of whether to forward the message or not. Sadly, most of contemporary query languages do not support stateful filtering. Note also that filters can reduce the

number of messages, but not their sizes. Given the sizes of documents exchanged in the Internet, distributed systems could benefit from components that perform not only filtering, but also processing of relevant messages. As a result they would produce new (presumably shorter and more specific) messages to be forwarded. The message processing components, which are one of key concepts of the framework presented in this thesis, could work on more than one stream of messages in order either to merge them, or to build composite messages.

From an application developer's point of view, the extension of message processing capabilities beyond filtering, offers an interesting option to delegate an important part of processing from end-user application (and machine) to the messaging subsystem (and remote machines). That allows constructing computation-intensive applications for relatively limited devices connected to the network. If appropriately designed, the application would not even require the user device to be connected permanently.

Dynamic deployment. Component placement is crucial to overall processing efficiency. When no prior knowledge of message source exists, i.e. when the system relies on message source discovery mechanisms, the only choice for a distributed system is to deploy the processing components dynamically after the sources are found. The nearer to the message source, the more effective (especially in terms of used bandwidth) the filter is. However, "near to the source" frequently means "outside the organization looking for message sources". Therefore, deployed components should not be run as standalone processes, but rather should be placed inside containers used to manage them. The proposed framework includes appropriate containers as well as other deployment-related mechanisms.

Opacity. The application developers' attention should be concentrated mostly upon their applications-defined protocols, not the underlying network. In order to hide the complexity of network communication from the applications (and developers), distributed systems are often implemented using middleware technologies. The use of middleware results in putting an abstraction layer upon the real network. The layer can be built upon a particular system developer proprietary solutions, but commonly it reuses generic middleware in order to facilitate future integration with other systems, possibly provided by different vendors.

When considering putting an abstraction layer on large heterogeneous networks, especially on the Internet, middleware environments constructing overlay networks with logical topologies resembling local area networks are of particular interest, because they substitute the programmer in dealing with many communication obstacles e.g. firewalls, various transport protocols etc. The family of peer-to-peer environments that have the described advantage has drawn much attention recently. REMP framework presented in this thesis reuses a peer-to-peer environment to enable the developers to easily create overlay networks dedicated to their applications.

To summarize, the key concepts of the proposed framework include:

- combining dynamic deployment and messaging services to allow the application developer to delegate message processing to the messaging subsystem and therefore to reduce the volume of messages transferred through the network,
- reusing (or building) middleware services to overcome communication obstacles such as firewalls, address translators, etc. to create a virtualized (overlay) network of needed instances,

- providing a flexible, semantics-based way of describing message sources and expressing queries regarding them,
- providing an easy to use, opaque service for creating application-specific overlay networks upon existing peer-to-peer environment.

By implementing the listed concepts in the middleware, the proposed framework opens a set of new possibilities to the peer-to-peer distributed systems developers.

This chapter outlines the area of application for the proposed framework for semantics based dynamic deployment of overlay networks and presents the research targets. Therefore, it is structured as follows. Section 1.1 introduces key definitions used in the thesis. Section 1.2 outlines the basic functionality required from the framework. Section 1.3 discusses the class of systems to be supported by the framework. Sections 1.4 contains the thesis and list of research targets for the presented work. Section 1.5 lists the original contribution of the author. Finally, section 1.6 presents the organization of the thesis.

1.1. Key definitions

In order to make the framework description clearer, the following definitions are used:

- **message processing framework** – a set of services that are used for construction of message processing systems,
- **message processing system** – an instance of the message processing framework, customized, deployed and configured, used by message producers and consumers,
- **message producer** – an actor (user application) that produces messages to be processed by the message processing system,
- **message consumer** – an actor (user application) that specifies the message processing overlay networks and receives the messages processed by the message processing system,
- **message source** – an object implemented by the producer application, the system interacts with; represents a single message stream of messages produced by the application,
- **message sink** – an object implemented by the consumer application, the system interacts with; receives a single message stream requested by the application,
- **message processing node** – a message processing system internal component responsible for processing of one or more message streams, typically to produce a new message stream,
- **message stream** – a set of messages described by a single description expressed in a meta-language, published sequentially by a single entity (source),
- **message processing (overlay) network** – an application-specific network specified by the user and implemented by the message processing system with a message distribution graph,
- **communication channel** – a logical link between message processing system entities used for sending and receiving messages; may be unidirectional or bi-directional, may have one or more sending endpoints and one or more receiving endpoints,
- **communication endpoint** – an object that implements either sending or receiving side of a transport protocol,
- **transport protocol** – a networking protocol that is used to pass messages between framework instance entities; not necessary the OSI model layer 4 protocol.

1.2. Overview of framework goals

The goal of the presented work is to design and partially implement a framework for creating peer-to-peer systems supporting dynamic deployment and maintenance of message processing overlay networks. The networks are expected to contain dynamically deployed nodes responsible for performing respective stages of processing.

There are two basic classes of actors for systems built with the use of the framework: message producers and message consumers. Message producers will supply the system with messages, while the consumers will submit their requests and receive requested messages. The actors are not expected to interact directly. Rather (as depicted in Fig. 1) they will interact with system internal entities.

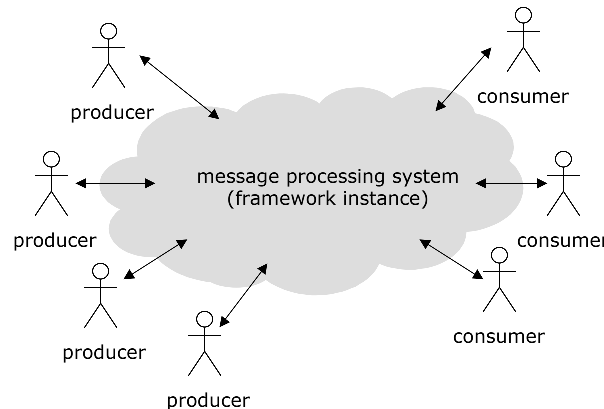


Fig. 1. The main actors of a system built upon the proposed framework.

The requests handling mechanism will allow the consumer not only to simply select the message source(s) of interest, but also to delegate a part of message processing code to the messaging subsystem. In order to support the delegation, requests are expected to carry specifications of message processing overlay networks needed to be created by the system, including descriptions of message sources and message processing stages. In order to create the requested networks in accordance with the specifications, the system will need to:

- search for producers capable of publishing messages that match consumers' queries,
- generate or find appropriate message processing nodes responsible for carrying out the delegated computations related to message processing,
- deploy the nodes and interlink them with communication channels.

The first two stages of network creation will be based solely on the message consumer's request contents, while the completion of the last will depend on current framework instance state also (it will be influenced e.g. by framework instance's reflection mechanisms). After completing the third stage, the overlay network created by system internals would act as a virtual message producer that creates messages exactly matching the consumer's request (Fig. 2). Because the overlay networks are supposed to carry out very specific computations, a one-to-one relationship between the number of consumer requests and created overlay networks is expected.

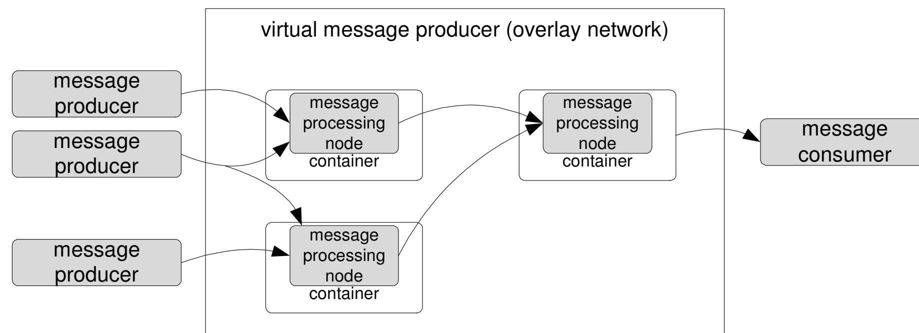


Fig. 2. The idea of an overlay network implementing the functionality of a virtual message producer.

In order to generate the message processing nodes, framework instances will need to translate the descriptions of message processing stages to deployable components. When deploying the components, the system would need to find appropriate (system internal) containers, which are the basic infrastructure building blocks.

The architecture of message processing systems created with the proposed framework will consist of three layers (Fig. 3):

- runtime environment layer,
- infrastructure layer,
- overlays layer.

The runtime environment and infrastructure layer will be configured and maintained by system administrator(s), while the overlays layer will be defined by message producers offering message publishing contracts and message consumers specifying the message processing overlay networks. The main task of framework instance will be to dynamically implement the specified overlay networks upon the current set of containers, i.e. to perform dynamic mapping between overlays and infrastructure layers.

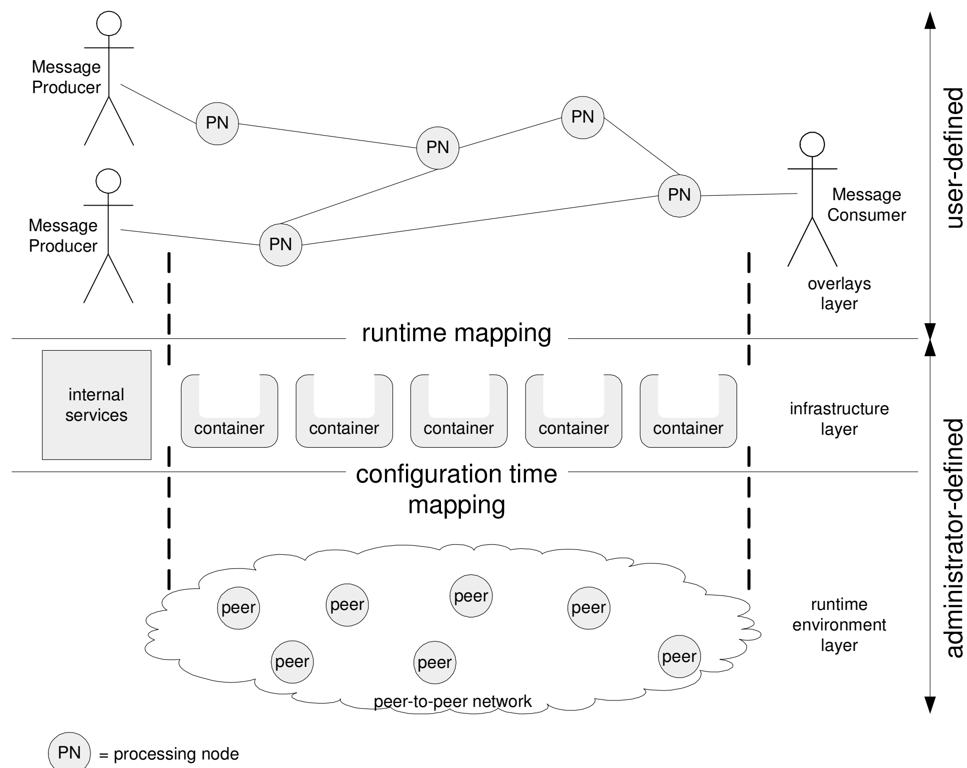


Fig. 3. Message processing system architecture overview.

After implementation, the framework instance will be required to maintain the networks. Among maintenance-related issues, assuring resiliency of the overlays seems to be one of the most important. For example, environmental changes, such as disconnection of a peer offering container's functionality, may force the system to rearrange affected overlay networks. In other words, the mapping between overlays and infrastructure layers could be performed many times even with regard to the same network. In order to be able to conduct the rearrangement, the system should be equipped with at least failure detection and recovery mechanisms, which should be implemented as framework internal services.

The concepts outlined in this section will be investigated further in Chapter 3, which deals with the analysis of requirements in order to create implementation-independent model for the framework instances.

1.3. Outline of supported message processing systems functionality

Depending on their design, middleware platforms are capable of supporting different classes of applications. REMP framework consists of a set of building blocks that provide a set of middleware services to the message processing systems developers.

As stated in section 1.2, a system built using the proposed framework, has to:

- search for appropriate message sources,
- generate components responsible for respective stages of delegated message processing,
- deploy and run the components in appropriate containers,
- link the components with message distribution graph.

This section briefly discusses the key options related to the outlined stages of overlay networks creation, the developers of the message processing systems are expected to choose from.

1.3.1. Selection of message sources and filtration of messages

A rough definition of the problem the middleware environments are dealing with is: *how to pass the information from one entity to another*? Because such definition is too general when it comes to implementation, the problem is narrowed to particular constraints that stem from the nature of service to be provided to the user of the middleware.

The communication channels, created to interconnect the communicating entities, are constructed either statically or dynamically (on demand). In order for the middleware to create a channel on demand, the message consumer's requirements (regarding the desired source of messages) must be expressed in some way. Therefore, from the message processing system designer point of view, the *set of criteria a message consumer can use to express its interest regarding message producers* is of highest importance. Depending on the way the consumers use to express their interest, three variants of the publish-subscribe pattern are commonly used:

- subscription for a stream of messages from particular source,
- subscription for a stream of messages on particular topic,
- subscription for a stream of messages with particular content.

The following paragraphs briefly characterize each of the listed variants.

Simple publish-subscribe. The most straightforward approach is known as simple publish-subscribe between two remote entities, typically represented as objects. In order to find appropriate message producer, the consumer has to know the reference (handle) of the producer. Such approach is simple and effective for systems that have well-defined structure and stable communication channels. Simple publish-subscribe systems may also provide means for specifying filters that would allow only the interesting (from the consumer's point of view) messages to be forwarded. Java Management Extensions (JMX) is just one of many popular environments that benefits from introducing such feature[Perry 2002][Kreger 2003].

Topic-based publish-subscribe. The topic-based publish-subscribe is more flexible in terms of communication channel stability, because the producers and consumers are interconnected by message passing intermediaries and therefore decoupled. Such approach is present e.g. by the Java Message Service (JMS)[Hapner 2002][Terry 2002]. In such case, the message passing system is partially aware of the meaning of the information passed through its channels. The meaning of the messages may be roughly defined by the topic assigned to a particular channel, the producers and consumers must agree upon. Therefore, the ability to search for appropriate message sources is based only on the messaging channel identifier.

Such approach is useful especially for systems that need communication channels between a few producers and many consumers. Moreover, if only the middleware provides a way to filter out uninteresting messages, the approach is quite flexible and useful not only for consumers interested in all messages that are produced and published to a particular topic-identified channel, but for the ones interested in a subset of such messages as well. Note however, that as the systems that use topic-based publish-subscribe evolve, the numbers of messages published to a particular topic may grow significantly. In order to reduce the number of unwanted messages, the developers of topic-based distributed systems may need to introduce changes to the addressing (the naming of messaging channels), and in turn to modify the user applications' code or configuration.

Content-based publish-subscribe. The content-based publish-subscribe systems allow their users to express their interest using predicates upon the contents of messages. The queries issued as a part of consumer registration process contain specifications of messages that are used by the middleware to identify appropriate message producers as well as the messages of interest. Such approach is much more powerful than the topic-based one, especially if the filtering functionality is desired. Placing filters close to the message producers can reduce the number of forwarded messages and therefore reduce the network bandwidth consumption. Moreover, if only the expressiveness of the query language allows, content-based publish/subscribe systems do not suffer from growing number of producers and consumers, because the applications express their requirements in terms of the contents of messages, which are immune to the systems' evolution. Therefore, the content-based publish-subscribe systems seem to be more scalable than the topic-based ones.

The REMP framework is designed in a way that reflects the needs of the content-based publish-subscribe systems and provides for extending its functionality e.g. by designing mechanisms for integrating their own message description and query languages. In order to provide maximum expressiveness, support for semantic message description and query languages is provided.

1.3.2. Message processing versus message passing

The envisioned instances of the REMP framework belong to the message processing systems class. In order to outline the set of applications the framework is expected to support, a clear distinction between message passing and message processing systems should be introduced.

Distributed data processing relies on passing messages between involved parties. Roughly speaking, a message is a piece of data transferred from a sender to a receiver, which may in turn act as a sender of another message. Typically, a message consists of two parts: payload provided by communicating parties and header (or trailer) that carries data meaningful for the message passing entities. The goal of the message passing systems is to quickly find an addressee of a message and to forward the message itself to that entity. Entities of message passing systems do not change the payload of a message and because of that, neither its syntax nor semantics is changed during the transport.

Message processing systems are able interpret not only the header, but the payload as well. In other words, entities of message processing systems are capable of consuming and producing messages. Note, that the processing may be reduced to reading and analyzing headers, so the class of message processing systems is a superset of message passing systems.

Allowing the message processing nodes to activate any time they need to, not only when a new message is received, is also an interesting feature. Having their own execution threads, the algorithms of processing are capable e.g. of gathering real-time context information. That allows for easy implementation of useful features, such as sending a processing report every hour, etc.

1.3.3. Component deployment

The term “deployment” is defined by the OMG as “the allocation of an artifact or artifact instance to a deployment target” [OMG-UML-S 2007]. The definition is generic and does not discriminate between static and dynamic deployment with regard to software components.

The overlay networks that are going to be created by instances of the proposed framework will consist of many *cooperating* components. In order to deploy such set of components dynamically, i.e. at runtime, a message processing system should:

- determine the set of possible deployment targets (i.e. containers capable of executing the component code) for respective components,
- select the containers for respective components by applying a deployment strategy (prepare a deployment plan),
- execute the deployment plan.

Because various framework instances (i.e. distributed systems creating overlay networks) are expected to have different deployment goals, the second of the listed tasks seems to be most instance-specific, and therefore can hardly be handled by the framework. The framework itself can only help developers by providing implementations for simple generic strategies and monitoring information for implementing more advanced ones.

Finding the set of available deployment targets (containers) is a common task and needs to be supported by the framework itself. Note that the containers are expected to reside on peer-to-peer network nodes. Because peer-to-peer environments may be dynamic

with regard to availability of peers, the task of providing dynamic discovery to the developers of instances is especially important. It gets quite complicated when the containers are not uniform, i.e. different configurations of the containers are allowed. In such case, the framework should be capable of matching the configurations of available containers against the requirements (dependencies) of particular components. That results in the need for introducing container configuration descriptors, component requirements descriptors and matching mechanisms.

The execution of a deployment plan is a common task also and therefore should also be performed by the framework mechanisms. The list of basic mechanisms includes downloading components' codes, executing them inside containers and linking the components together with communication channels. Note however, that in order to make the container host safe from malicious components, advanced mechanisms such as controlling components' access to the host services should be taken into consideration.

1.3.4. Inter-component communication

In order to perform any useful computation, message processing nodes need to be interlinked with communication channels used to forward messages from their sources to respective destinations. However, to make the deployment easier, the communication details should be opaque to the nodes. A common way of achieving that goal is to separate the communication from the processing code by externalizing the descriptions of communication endpoints. The party responsible for creating specification of the message processing overlay (i.e. the message consumer) is expected to include the specification of all communication endpoints. A component descriptor, that contains the specification of the inbound and outbound endpoints, is interpreted by runtime environment at component deployment and startup time. The approach chosen for the REMP framework follows the same pattern.

From the message processing overlay logic point of view, the communication channels are spanned between message processing nodes. However, as the nodes are implemented as components residing inside appropriate containers, the channels in fact are spanned between the containers, which act as intermediaries between communicating nodes. Therefore, in order to make the components residing in the same container distinctive, an appropriate addressing scheme needs to be designed. Moreover, in order to allow for interconnections between overlay networks, the addressing should provide system-wide unique addresses to the components. In order to be able to create communication endpoints, the overlay networks components addresses should be resolvable to the containers' ones.

In order to support as broad range of applications as possible, imposing limits on numbers of sending and receiving endpoints of communication channels should be avoided. In an ideal case, a communication channel should be a logical entity, created and destroyed neither by the senders nor by the receivers. Such approach would allow the message processing system not to bother with providing special treatment e.g. for a node that created the channel, but is no longer interested in using it. Reusing and adapting an existing solution or designing and implementing own method for implementing such logical channels seems to be one of key task for the framework.

1.4. The thesis and research goals

Given the described considerations, the thesis of presented work is: *contemporary peer-to-peer middleware environments provide enough support for building a framework for dynamic deployment of message processing overlay networks. Augmentation of peer-to-peer middleware with semantic descriptions of messages improves the flexibility of the message processing delegation service.*

In order to prove the thesis, the main goals of the presented work are:

- *to design an architecture of a general-purpose framework for constructing message processing systems using dynamically created application-specific overlay networks,*
- *to embed matching between semantic descriptions of messages the message producers are able to provide and queries of message consumers in the framework,*
- *to construct mechanisms for easy configuring, reconfiguring and extending framework instances,*
- *to implement the core of the framework upon an existing peer-to-peer technology.*

1.5. Original achievements

The presented work is based on an original approach to opening the peer-to-peer environments for general purpose, component-based message processing applications. The main idea that motivated the presented work was to use the resources available in peer-to-peer networks to implement an environment for executing complex applications with devices limited in terms of available memory, processing power and network bandwidth. In order to achieve the goal, a series of original contributions, were made. From the author’s point of view, most important of them are as follows:

- formulation of a concept of implementing component-based message processing applications as mutually independent overlay networks; contrary to most of contemporary distributed systems, elements of the networks are created at runtime, in a way that reflects the properties of the runtime environment,
- formulation of a concept of integrating the dynamic generation and deployment of application components into peer-to-peer concept-based publish/subscribe facilities,
- prototype implementation of a framework that provides an application execution environment in order to verify the aforementioned concepts in practice,
- introduction of a partially-declarative model of peer-to-peer application development; the model allows for declarative specification of desired properties of the containers to be used as well as of the inter-component communication channels,
- design and implementation of message semantics driven dynamic deployment service; results of semantically expressed queries are the most important criteria that drives the selection of containers to be used by the deployed overlay network,
- design and implementation of solutions that support collaboration between applications by enabling them to reuse their intermediary processing results,
- practical evaluation of the proposed concepts.

1.6. Organization of the thesis

In order to meet the specified targets the presented work consists of:

- discussion of key design problems related to the framework in context of existing technologies and solutions (Chapter 2),
- specification of implementation platform independent model of the framework instances (Chapter 3),
- discussion of key issues related to the implementation of framework core (Chapter 4),
- evaluation of prototype implementations of core services (Chapter 5).
- summary and concluding remarks (Chapter 6).

2. Survey of related middleware environments

Support for development of large scale, heterogeneous distributed systems has been a popular research topic from a long time. As a result, many middleware technologies have been developed. The diversity of contemporary distributed systems' applications and runtime environments is to some extent reflected by the diversity of middleware technologies[Mahmoud 2004]. Most of the differences result from choosing various solutions to the common design issues. In case of message-oriented middleware class the list of the issues includes design or choice of:

- processing models,
- mechanisms for distributing and interpreting metadata,
- schemes for addressing distributed system entities.

The goal of this chapter is to survey the existing middleware technologies (both message-oriented and not) in order to find solutions for the outlined issues that could be reused in the framework as well as to find non-functional features that could help the developers of framework instances. Section 2.1 presents popular models of message forwarding and processing. Section 2.2 surveys the solutions for distributing metadata regarding semantics and syntax of messages as well as system services. Section 2.3 discusses the abstractions, such as addressing schemes that help to virtualize the execution environment. Section 2.4 concentrates upon services provided by various environments that could be used internally by REMP framework instances to adapt to the runtime environment changes. Finally, section 2.5 summarizes the survey.

2.1. Key concepts of existing message passing and processing systems

Contemporary message passing and processing systems are frequently based on shared infrastructure built by message brokers (rather than on centralized message servers). In order not to make the functionality offered by the proposed framework instances more constrained comparing to the existing systems and reuse some of their ideas, key of their characteristics are outlined in the following sections.

Section 2.1.1 discusses the models of message passing implemented by contemporary systems, while 2.1.2 overviews the common entities and algorithms used to route the messages. 2.1.3 focuses on available message processing options and 2.1.4 discusses briefly quality of service parameters regarding message passing. Section 2.1.5 summarizes the overview and points out the ideas to be reused (either directly or not) by the framework.

2.1.1. Message stream processing models

There are two main message stream processing models implemented by contemporary middleware: channel-based and graph-based. Although they can be implemented in various ways that may affect the final usability, key characteristics important for the middleware user result from the logical organization of processing. Therefore, this section reviews the key attributes of the models in order to choose a base for logical topology to be used by the proposed framework instances.

Channel-based processing

Simple messaging services, such as CORBA Event Service[OMG 2000-1] introduced the concept of event channel as an intermediary in order to decouple message producers and consumers. The concept (although in a much mode sophisticated form) is constantly reused in integration technologies [SAS 2008], either directly, or as a convenient abstraction of the underlying mechanisms' complexity.

The simple, channel-based approach works fine only if all messages produced by producers are relevant to all consumers. In other case, message filtering messages should be used[Hohpe 2003]. Functionality of filters is defined by rules specified by message consumers. In simplest case, filtering is performed at consumer's host. Such approach provides for maximum flexibility, because no restrictions are put on filter designer, but does save neither network bandwidth (all messages are transported to the consumer host), nor computing power of the consumers machine (all messages are still processed by the machine).

In order to reduce the bandwidth consumption and computing power, the filters must be delegated outside the consumer's machine. There are a number of services that support such functionality. For example, the lack of filters in CORBA Event Service was addressed by the CORBA Notification Service [OMG 2004]. The Notification Service provides event filtering by introducing a content-based filtering language, based on CORBA Trader Constraint Language (TCL), specified by the CORBA Trading Service [OMG 2000-2]. Each of the constraints a filter consists of, is defined as a sequence of names of types the constraint is applicable for and a string containing boolean expression in extended TCL [Gore 2001].

The flow of messages in CORBA Notification Service is depicted in Fig. 4. Messages on their way from producers to consumers may undergo multiple filtration operations, because many parties, i.e. message producers, consumers and event channel administrators are provided with means for attaching message filters. Message producers (called suppliers in CORBA terminology) submit their messages to proxy consumers, which act as gateways for event channels. The first stage of filtering occurs at the proxies that are created on a per-supplier basis. It is worth to note that both pull and push patterns are supported by the event channels through the use of pattern-specific proxies. The proxies themselves are created and managed by "admin" objects that also perform message filtering on a per-supplier-group basis. The consumers' side is designed in a similar way – each consumer is connected to a filtering proxy supplier. Proxy suppliers are created by consumer-side admin objects. Support for multiple admin objects allows for logical grouping of suppliers and consumers.

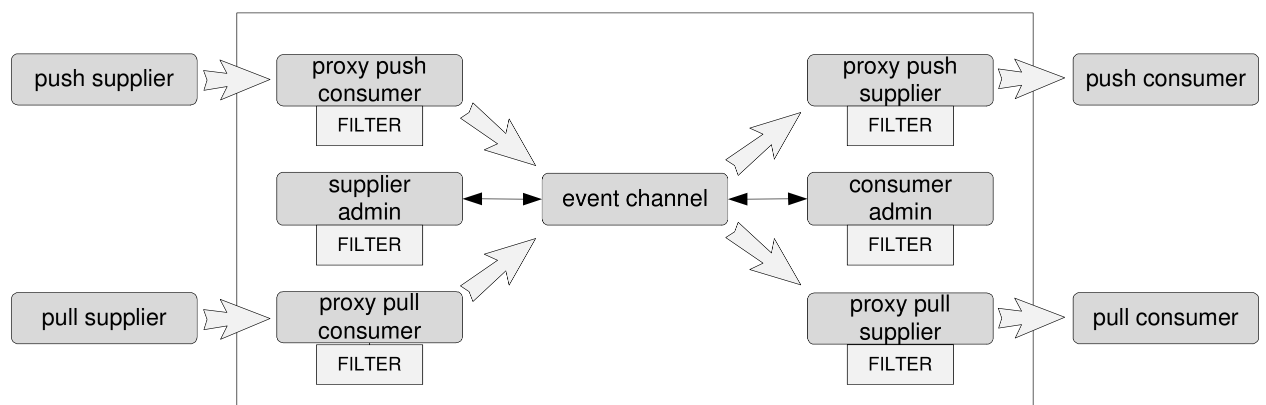


Fig. 4. Message flow in the CORBA Notification Service.

The scopes of filters' influence depend from their placement. Messages discarded by filters placed at the event channel entry are not forwarded to any of the consumers, while the ones discarded at the exit are not forwarded only to the selected consumer. On the other hand, the sooner an irrelevant message is discarded, the better e.g. in terms of bandwidth consumption. In order not to discard important messages while filtering out unnecessary ones and placing filters close to the message sources a messaging system developer may use an expressive filter definition language that performs deep analysis of the message contents.

Graph-based processing

Message oriented middleware environments enable inter-operation of large, heterogenous and dynamically changing distributed systems. The environments are popular in industry – there are many products that provide message passing, including IBM MQSeries (later IBM WebSphere MQ)[Davies 2005], Microsoft Message Queue (MSMQ)[Redkar 2004], TIBCO's Rendezvous [TIBCO 2007] and many others.

IBM WebSphere MQ is now one of the leading message oriented middleware platforms. One of its key concepts, federated brokers comes from an earlier project, Gryphon[Banavar 1999-1], which started in 1997 and proved to be scalable by serving more than 100000 worldwide concurrently connected clients during the Olympic Games and other sports events.

Gryphon introduced a concept of an information flow graph (IFG), which consists of:

- information spaces, which are either event histories or event states, described by a message schema, and which are the nodes of the IFG,
- data flows, which form the arcs of the IFG, and which have an associated operation, such as "select", "transform", "collapse" or "expand".

A sample IFG is depicted in Fig. 5.

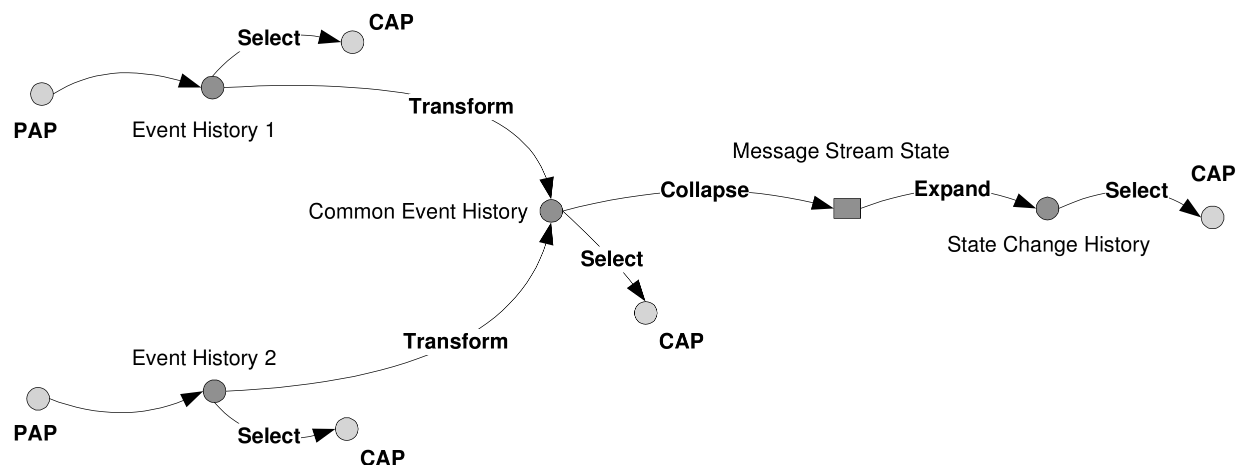


Fig. 5. An example of Gryphon's information flow graph
(PAP – publisher access point, CAP – consumer access point).

The event histories are a either ordered or unordered collections of messages, which grow as the messages regarding particular events are published. Event histories are needed in order to assure the retransmission in case of failure in message delivery [Bhola 2002]. Message producers and consumers are represented in the IFG also as event histories. Multiple event histories may exist in an information flow graph; a client application can

connect to any of them if the results of the processing performed by earlier operations are of interest to the application.

Event states are IFG nodes which perform operations on data flows, e.g. hold the current maximum of particular value. They cannot be used for message retransmission, but can be used by intermittently connected message consumers which are interested only in the state of the flow, not in all the messages that form it.

The “select” operation is defined as an arc connecting two histories of the same schema. A predicate on the attributes of the message type is used to determine whether a particular message will be forwarded from one history to another or not. In order to connect histories described by different schemas, a “transform” arc is used. The arc has an associated rule for transforming the syntax of messages. The transformations are used e.g. in order to combine histories of syntactically different, but semantically equivalent messages.

The “collapse” operation is defined as an arc connecting an event history to an event state. The rule associated with the arc transforms a current state and a new event into a new state. As an example, consider a state that holds an average value of last ten observations of some variable. Whenever a new message comes, the average must be adjusted accordingly. In order for the user to be able to keep track of the changes of a particular event state, the “expand” arc is defined. Such arcs link an event state to an event history. Whenever the state changes, the history is sent a new message.

The logical information flow graphs are implemented upon a network of content-based routing brokers. The brokers take care of all crucial functions of the Gryphon middleware, including managing subscriptions, matching consumers requests against schema that describe publishers’ information spaces, processing the information flows (performing requested operations) and delivering messages to the consumers. The brokers used in Gryphon interpret a dedicated message processing language. Because scalability is the main design goal, its expressiveness is limited to logical and arithmetical operations on messages defined by their schemas. Embedding the interpretation of message streams in brokers result in improvement of not only system efficiency but also quality of service. For example, the ability to interpret the message streams gives brokers a chance to detect out-of-order messages and compensate the changes in event states accordingly [Zhao 2001].

Filtering in Gryphon is based on matching the predicates regarding the contents of messages against the messages themselves. The matching mechanism is based on the message syntax. The syntax of messages stored by each information space is described by a schema defined upon a simple set of atomic types [Sturman 1998].

Gryphon’s IFG is interesting from at least one reason: operations on messages are not performed only by graph nodes, but by arcs as well. Therefore, it is not possible to implement it using a simple “node=component, arc=connection” pattern. Rather, the graph must undergo a set of transformations to be implementable in the brokers network. Because the main design goal for Gryphon is scalability and the key for scalability is the ability to filter the message contents close to their sources, the graphs are reorganized so that the select operations move as close as possible to the source and the transforms are moved closer to the consumers. Such approach leads to shortening the paths present in the graph (in terms of hop count). On the other hand, the number of the paths becomes greater. The intermediate information spaces of any graph can be reused by other graphs. In order to support that, additional nodes are placed in the graph after reorganizing [Banavar 1999-3]. The process of carrying out operations necessary for graph implementation is called “graph rewriting”.

The overlay network is created by the brokers which use inter-broker protocol upon transport protocols. The broker network uses simple flooding algorithm with regard to subscriptions [Baldoni 2003]. That does not compromise the overall effectiveness of the network as long as the number of subscriptions is relatively low compared to the number of processed events.

The subscription and event transport mechanisms are based on the TCP/IP protocol stack. For example, IP multicast (both unreliable and reliable) is used in order to reduce bandwidth requirements [Banavar 1999-2]. Moreover, in order to support reuse of existing information flow graphs, introspection mechanisms are also supported by the brokers.

2.1.2. Message routing

Application layer routing of messages is conceptually similar to its network layer counterpart regarding IP packets. Message passing systems usually distribute the message passing functionality among a set of redundantly connected forwarding entities, called brokers. A broker is a general-purpose application layer router and processor of messages. Message forwarding principles are similar to those of IP multicast routing[Deering 1990]. The task of each broker is to process a subset of messages forwarded through the messaging service and to forward them in appropriate directions (either to the end receivers or next-hop brokers). In order to receive messages, a message consumer needs to register its interest in one of the brokers. Through the use of inter-broker protocols, the events of a broker joining/leaving the network may be managed almost transparently to the end user [Baldoni 2004][Zhou 2006].

There are many algorithms of application layer routing – starting from the simplest flooding-based approach, through more complex reference or topic-based, to complex, content-based ones that might use not only sink subscriptions, but also source advertisements to optimize message forwarding. This section briefly overviews the techniques.

<u>Flooding versus “smart push”</u>

Flooding is the most straightforward technique that ensures that every interested message consumer is delivered the requested message. Each message broker forwards each new message to all of its neighbors, save the one the message was received from as well as all of its registered consumers. Because all the messages are eventually processed and forwarded by all brokers, each of interested consumers will receive it. Pure flooding based approach can be implemented on a set of tree-shaped directed graphs with their roots at message producers and leaves at message consumers. However, because in such case the brokers do not filter messages, flooding may result in waste of bandwidth and processing power (Fig. 6). While in small scale, local area systems that could be acceptable, large scale systems cannot afford that. Moreover, system operating on directed graph is unable to provide the message producers with feedback information, and therefore cannot assure e.g. reliable delivery through retransmission of lost messages.

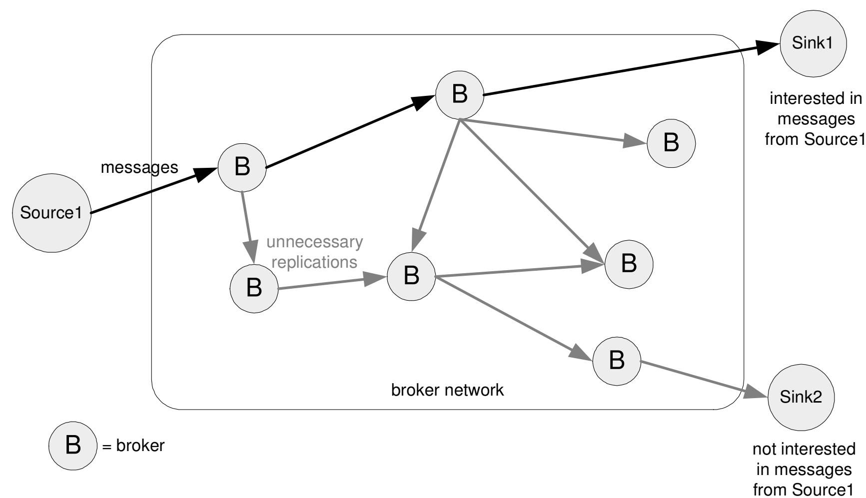


Fig. 6. Waste of resources in a pure flooding-based approach (example).

As shown in [Denning 2006], changing the "push" scheme of delivery to "smart push", in which the system delivers only messages which are needed by their consumers can result in dramatic improvement in terms of bandwidth usage and system scalability while not sacrificing user convenience. In order not to waste bandwidth and processing power, "smart pushing" systems require the message consumers to describe (register) their interest. The information gathered at message consumer registration time is used to configure a forwarding path in the network.

Contrary to the flooding based approach, when smart push is in effect, messages are forwarded only through configured forwarding paths. There are many points in which the overlay networks resemble the "traditional" IP-based ones. Similarly to IP routers, brokers maintain constantly monitored and updated routing tables. However, the contents of the tables may provide for more sophisticated routing table optimization algorithms.

In the case of reference-based addressing and routing, there is not much place for innovations with regard to message forwarding, because the information carried in addresses is much the same as in IP. In overlay networks which address their nodes only by assigning them ordinary numbers, addresses carry little information. Although, in terms of the OSI networking model, the routing based on such addresses is considered application layer routing, the decisions taken by message forwarding entities are based on information similar to the network layer routing. The size of routing tables depends mainly on the organization of the address space. In case of flat address space, when the identifiers of network entities have no structure, the routing tables tend to be large. If any hierarchy is introduced, the routing table entries may be aggregated. The decrease in routing table size is paid with reducing the particular router's network topology knowledge granularity. The tradeoff is the same as in the "traditional" networks.

When topic-based routing is used, any network node that declares ability to produce or consume messages related to a particular topic, not only declares its membership in particular logical domain, but also denotes its role in the network – producer or consumer of messages. Distributed infrastructure that supports the routing of messages between producers and consumers and functions similarly to network-level multicast routers attaches such node to a root or leaf of respective multicast tree. Topics introduce a bit of semantics to the message passing systems. However, as the topic name is the only manifestation of the semantics of messages, processing of the meta-information is very limited, because the set

of relations between topics is very limited. Commonly the messaging systems support only aggregation of topics making the applications able to subscribe for “all subtopics of …”.

In content-based routing, not only the roles of network entities are distinct, but also the message forwarders interpret the contents of messages. Because the routers create multicast trees based on individual subscriptions, the subscriptions themselves must carry information regarding contents of interesting messages. There are a few techniques, based on subscriptions, that are used to optimize the flows of messages. The following section reviews the techniques.

Content-based routing algorithms

This section reviews the routing mechanisms that are used in contemporary message passing systems in order to determine the concepts and kinds of metadata messages that might be reused by the REMP framework.

Filtration-based routing. A simple approach to route content-addressed messages derives from the transparent LAN switching concept. Each broker maintains a subscription table, which is constantly updated as new subscriptions are registered or canceled. The subscription messages are flooded through the network of brokers in order to assure each of them has an entry regarding the subscription. Because the subscription is flooded from the message consumer to the producer (i.e. backward to the potential message stream), the forwarding databases can be filled with regard both to the subscription (index) and forwarding direction (value). Whenever a message comes to the broker, it is matched against its subscription table and forwarded only in assigned directions (Fig. 7).

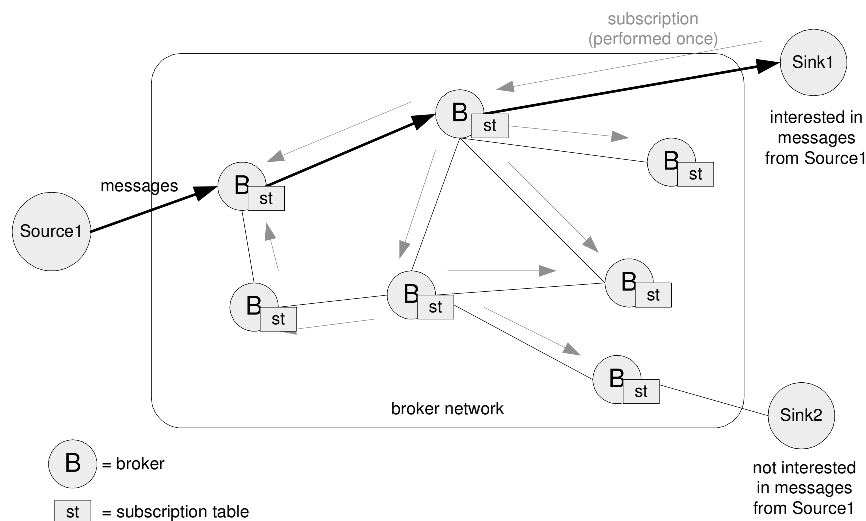


Fig. 7. Filtration-based routing (example).

The primary virtue of such approach is the reduced number of messages forwarded through the network, because brokers forward messages only to selected neighbors. Another advantage is the possibility of finding alternative directions for messages in case of broker disconnection. Based on query/response algorithms, similar to the ones implemented in IP dynamic routing protocols, the broker network can be made capable of repairing the broken flows.

The main drawback of such approach is quite straightforward. While IP routing effectiveness relies heavily on address aggregation and hierarchical organization of addressing, the simple content-based routing does not. In the simple approach, the content-based address space is considered flat, i.e. the subscriptions do not undergo any

cross-analysis at brokers. That results in large sizes of subscription tables. Moreover, because all the brokers need to be notified about all subscription changes that occur in the network, in dynamic systems the tables tend to be modified very frequently.

Simple content-based routing scheme is used e.g. in Gryphon, a publish/subscribe middleware created by IBM and aimed at distributing large volumes of data in real-time to thousands of clients distributed throughout a large public network.

Covering-based subscription routing. Keeping all the subscriptions present in a network on every broker resembles keeping all the addresses of single hosts in every IP router's routing table. Just as in the case of IP routing, content-based routers frequently do not need such detailed information in order to forward messages effectively. Carzaniga [Carzaniga 2004] has introduced an alternative to the simple routing scheme. The algorithms used in SIENA and JEDI (Java Event-based Distributed Infrastructure) [Cugola 2002] optimize not only the message forwarding, but subscription forwarding as well. In short, during subscription forwarding the brokers test if the newly coming subscription is not covered by a previously processed (and forwarded) one. If that is the case, the new (specific) subscription is not forwarded in the same directions. That results in most general subscriptions forwarded farther, while more specific ones stopped nearer to the message consumer (Fig. 8).

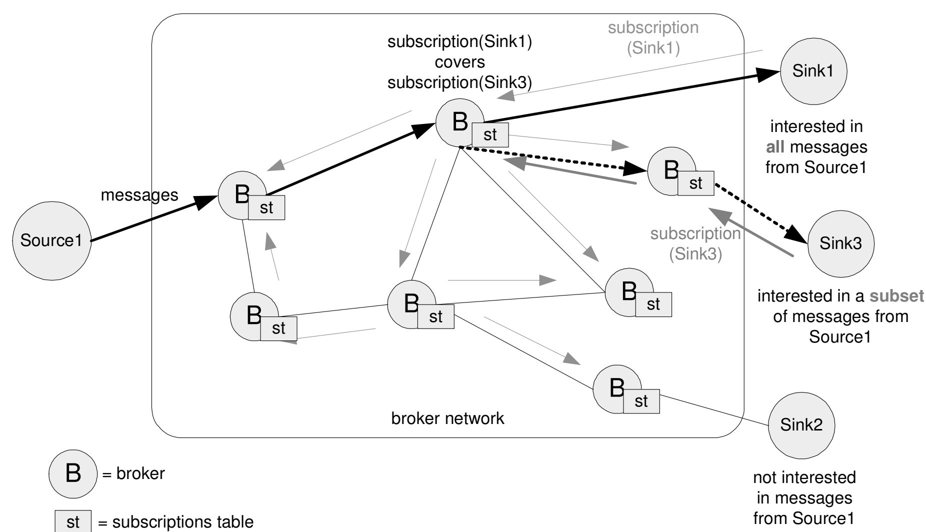


Fig. 8. Covering-based subscription routing (example).

In comparison to the simple approach, covering-based routing results in reducing the overall broker processing overhead and downsizing the subscription tables. In covering-based routed networks, brokers typically do not have complete knowledge about all subscriptions registered in the system. In a way, the covering test serves as an aggregator for subscriptions. The covering relation clusters the set of consumers, so that the addressing scheme no longer resembles flat address space. That reduces not only the overhead related to forwarding of subscriptions, but also provides for faster filtration of forwarded messages.

The drawback of this approach is more processing needed in case of the general subscription cancellation. Because the general subscriptions are not dynamically generated by brokers, once the consumer that registered the subscription cancels it, the record must be removed from all the brokers that kept it in their subscription table. That result also in the need of re-processing the more specific subscriptions covered by the removed one.

Source advertisements-aided subscription routing. The forwarding of subscriptions can be optimized with the use of advertisements of message sources. Such advertisements

can be defined as filters issued by message producers to indicate their intention to publish notifications [Mühl 2002][Carzaniga 1998]. Advertisements are published by potential message sources upon their registration. Aside from the message routing tables built from the active subscriptions, brokers keep subscriptions routing tables, built from the active message source advertisements. The tables are maintained using similar algorithms with regard to forwarding new and canceled advertisements through the broker network. When source advertisements are in use, subscriptions can be matched against them and forwarded only in the directions of potential sources (Fig. 9).

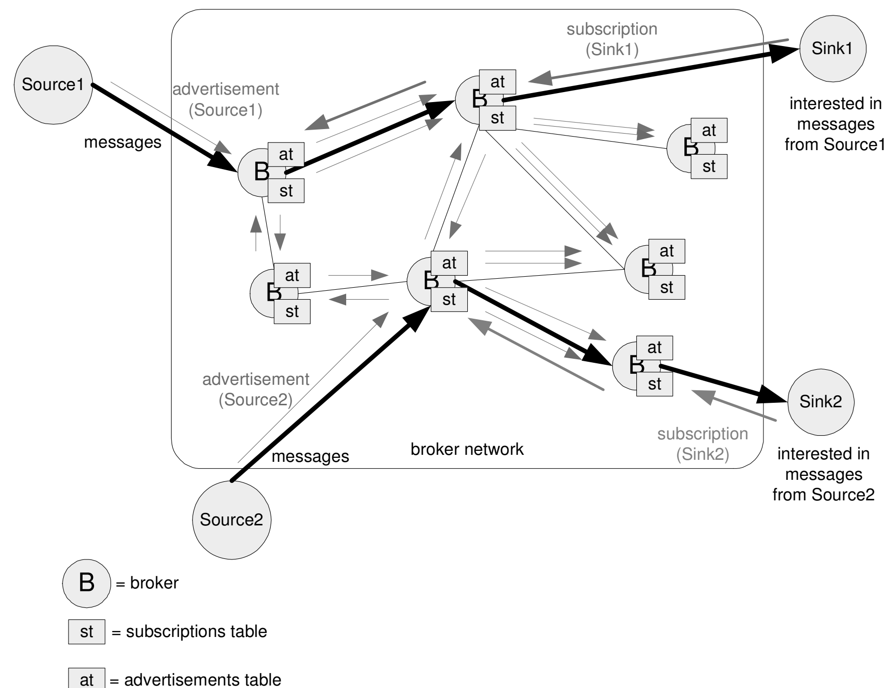


Fig. 9. Subscription forwarding based on source advertisements (example).

Source advertisement aided subscription routing may be beneficial especially for systems that serve many subscriptions, which happens when the number of sinks is high in comparison with the number of sources or when the sinks join and leave the system frequently.

2.1.3. Message processing and reuse of intermediary results

Processing of messages typically occurs at the message consumers and consists of three steps:

- selection of relevant messages,
- processing of messages' contents,
- change of consumer state.

Message passing systems frequently enable the message consumer to delegate the first stage of processing i.e. to define filters to be placed close to the message senders[FUSE 2009]. The filters are submitted to the system at message consumer's registration time. Depending on the system, they may undergo processing or be forwarded without any change. It is worth to note that the message consumer does not need to know, which scenario is in place.

Appropriate filter placement is crucial for message passing optimization. Some of the message passing systems, e.g. CORBA Notification Service, leave the decision regarding the placement to their users and administrators, while some place the filters automatically. In

the second case, the place of filter deployment may be unknown to the message consumer that defined the filter. Such opacity allows the developers and administrators to employ various filter placement strategies without requiring the users to adjust their applications. For example, in the case of Gryphon, the processing graphs are "rewritten" before deployment, i.e. the processing operations are moved towards the message producers in order for most of the brokers to perform only routing, not processing of messages.

Various languages are used to define message filters. Some systems adapt syntax used in other applications (SIENA[Carzaniga 2001], CORBA Notification Service, Elvin[Elvin-sub]), while others define their own (Gryphon). Noteworthy, each of the systems uses only one message filter definition language.

Because filters are typically based on message contents, the message consumers need to have some knowledge regarding the contents before submitting their query. The knowledge of message contents is typically carried outside the message passing system, e.g. in message producer documentation. However, if only introspection mechanisms were provided, the message consumers would be able to gather the information directly from the system. In large-scale systems such functionality seems desirable. As a basic mechanism supporting introspection, message source advertisements should be considered. Source advertisements may be useful not only for message brokers, but also for consumers who want to retrieve information about "what is published in the network". Some of the existing environments, like Elvin[Segall 2000] or SIENA, support source advertisements for that purpose.

Support for processing messages and reusing intermediary (partially processed) messages makes the environment more powerful at the cost of introducing more complexity to processing chain maintenance. Because the message payloads are processed, their semantics and syntax may be different from original. Existing environments, like Gryphon, allow for delegating message processing outside the consumer application and for attaching consumers at certain points of processing chain (see Fig. 5). However, because efficiency is the main design goal for most of the environments, the spectrum of operations allowed outside message consumer application is limited.

2.1.4. Quality of service

Quality of service provided by an overlay network depends heavily on the underlay network stability. Because in case of Internet-based overlays there are little possibilities for the overlay to manage the underlay, the list of quality of service parameters is limited. What may seem surprising even assuring basic parameters, such as reliability and ordered delivery is, in case of message passing overlays, a non-trivial task.

CORBA Notification Service provides its users with one of the longest list of quality of service parameters. When using that platform, quality of service properties can be set at various scope levels, corresponding to notification channel, supplier/consumer group, proxy supplier or consumer and individual message (in the variable part of message header). The message channel level (most interesting from the proposed framework point of view) defines attributes such as [Bartlett 2001]:

- reliable delivery,
- discard policy,
- message ordering,
- message prioritization.

End-to-end reliability seems to be the hardest to implement. Taking into account that the messages are processed on their way to destinations, in order to recover a lost message, a message passing system must be able to analyze the message flow in backward direction. Solutions for ensuring reliability are based either on a persistent storage attached to message publishers and acknowledgements from the receivers [Bhola 2002], or on distributing digests of messages among brokers [Costa 2004]. The developers of the Elvin[Segall 2000] system deliberately resigned from assuring reliable delivery because of scalability problems.

Because the proposed framework works on deployment targets level, none of the approaches may be reused to provide end-to-end reliability. On the other hand, hop-by-hop reliability can be implemented quite easily (by using simple acknowledgements).

A discard policy can be understood as a basic filtering mechanism. What makes the mechanisms different, is the source of rules that build them up. While message filters are defined by the receivers, the discard policies are defined by the administrator of message producer. The most straightforward discard policy is “drop a message if there is no receiver that requests it”. Because the proposed framework is expected to make use of message channels, and a message channel is fully set up when at least one sender and at least one receiver are attached, such policy can be implemented very easily. Other policies may also be defined, based e.g. on messages content or message expiration time.

Ordered delivery of messages can be implemented by middleware simply by tagging the messages with sequence numbers and buffering the newer messages until the older ones are delivered. Note that the buffering makes sense only if reliable delivery is also guaranteed. Otherwise, the buffer needs to have a timeout set for each of the missing messages in order not to stop the whole service due to one lost packet. Ordered delivery is provided e.g. by Gryphon and JEDI[Cugola 2001] systems.

Prioritization makes sense when a service is expected to suffer from bottlenecks, which is common case for services deployed over Internet. There are many queuing algorithms developed that support prioritization. However, because the message distribution graph channels are expected to carry messages of single type, it is questionable, whether advanced queuing techniques are really needed. It seems that simple prioritization will satisfy the users. The Elvin message passing system reuses also a concept of quenching sources that issue too many messages [Segall 2000]. Because quenching in case of message processing system needs backward analysis, simple adoption of the concept seems problematic. However, means for either for reducing the volume of messages published by “too talkative” sources or for subscribing for a subset of message stream must be considered.

2.1.5. Choices for the proposed framework

Most of contemporary middleware platforms implement channel-based or graph-based models of message processing. Typically they allow users only to filter the message streams in order to reduce the overall bandwidth requirements in exchange for a relatively low additional computing power overhead. Additionally, some of them provide simple, syntax-based, stateless message transformation mechanisms.

From the two reviewed models, graph-based, although harder to implement and maintain, offers much more flexibility to the designer of a distributed system. Therefore, it seems that the graph-based model will suit the proposed framework better. The framework introduces more sophisticated functionality than simple filtering by extending the delegated

functionality to stateful processing of messages. Therefore, simple adoption of Gryphon's IFG is not possible. Moreover, because the processing semantics is going to be hidden in the message processing nodes, and therefore the graph cannot be "rewritten", as in the Gryphon's case, labeling of arcs similarly to IFG seems to be useless.

Contemporary message passing and processing systems create a shared infrastructure of brokers. The proposed framework goes one level of abstraction deeper, i.e. creates a shared infrastructure of containers. Therefore, none of the systems based on broker networks can be directly reused to create the framework. Nonetheless, shared infrastructure of brokers can be created upon shared infrastructure of containers by implementing a broker as a deployable component.

The general-purpose brokers form non-directed graphs. In order to support creating a message processing system based on brokers as a message processing overlay network, two-way logical connections between neighboring entities should be provided. In order to satisfy the developer, the framework should support as wide range of message stream selection options as possible. Support for content-based subscriptions expressed by semantic message descriptions is very desirable.

Message passing and processing systems functions are driven by the queries issued by message consumers. The user-provided specification of interesting traffic has two main functions:

- specification of the message stream(s) the user is interested in,
- specification of desired operations on messages that belong to the selected stream(s).

It seems reasonable that the functionality of message processing overlay network is defined in the same way.

In order for "smart push" to work efficiently, message processing systems place message stream filters and processors near to the message sources. In most cases, the filters and processors are stateless, i.e. they are expressed as scripts executed upon each received message. Introducing stateful components that e.g. analyze not only the message itself, but also looks at the context information, defined e.g. by previous messages would increase framework's flexibility.

Depending on the application, a developer may decide to delegate various message processing code to a message processing system. In order to support a broad range of applications, an expressive programming language for defining functionalities of message processing nodes needs to be chosen. However, because the semantics and syntax of exchanged messages are the only common properties of neighboring message processing nodes, it does not seem necessary to limit the developer to use a single programming language. Rather, the framework should facilitate development of multi-language message processing systems.

While the created overlay networks are expected to serve various purposes, the intermediary processing results might seem usable outside a particular network. Therefore, message syntax and semantics externalization mechanisms availability should not be limited to the actors of message processing systems. On the other hand, it seems a waste of resources to externalize the metadata at every step of processing. As a compromise, the overlay network designer should be provided with means for externalizing the metadata at selected points. Such concept resembles the consumer access points of Gryphon's IFG, save that Gryphon creates the access points at each event history.

Allowing reuse of intermediary results makes the computations performed by a message processing system's internal entities more complex. At a first glance, two of them seem most obvious. First, there is a question whether the results receiver network influences the deployment plan for the producer network or not. Second issue regards the lifecycle of the producer overlay network – should the receiver network keep a part or whole producer network alive after the message consumer that created the network closes the network down, or not? If so, will it be allowed to re-deploy the part of the producer network that is kept alive in order to better serve the consumer network? Issues like that regard more aspects of the intermediary results producer network, in particular its security.

Messages are not the only intermediary processing results that may be reused. What can also be reused (either for decent purposes or not) is the code of message processing nodes. General-purpose message processing nodes, e.g. message mailboxes, can be reused for many types of messages. The usage of such off-the-shelf components can shorten the time of overlay networks development and provide for greater reliability, given that the components are more intensively tested. Therefore, a solution for components storage should also be considered.

Contemporary systems sometimes make use of automatic optimization of consumer queries. Although it is hardly possible to design an optimizer for an extensible multi-language environment, interfaces for adding query optimizers should be exposed.

Implementation of end-to-end quality of service parameters in a message processing system is hardly possible at the container level. Nonetheless, providing hop-by-hop QoS seems to be feasible. In particular, in order not to forward too many messages, the message producers should advertise not only the semantics and syntax of produced messages, but contract options regarding e.g. minimum and maximum intervals between subsequent messages. Such simple enhancement would facilitate bandwidth saving if only message consumers use such option in a reasonable way.

2.2. Metadata representation and distribution

The term „metadata” is commonly understood as „data about data”. Metadata is the data describing context, content and structure of data records as well as their management.

Metadata is the basis for middleware computations, just as the exchanged data for distributed systems implemented upon the middleware. In context of middleware environments, the term “metadata” may in particular refer to:

- semantics and syntax of exchanged data,
- functioning of middleware internals.

The goal of the survey presented in this section is to select solutions to represent metadata regarding the syntax and semantics of messages in the message processing systems built upon the proposed framework. The survey starts with discussion of type systems externalization (section 2.2.1). Then message syntax and placement of metadata related to particular messages are discussed (subsection 2.2.2). Following subsections (2.2.3 and 2.2.4) overview the standard languages for representing metadata related to message syntax and semantics, respectively. The final subsection (2.2.5) summarizes the discussion and presents choices for the framework.

2.2.1. Externalization of type systems

Type systems are lightweight formal methods to ensure that a program expressed in the language behaves correctly with respect to its specification. Programming language designers use them more or less formally to classify the phrases written in the languages in accordance with the types of values they compute[Pierce 2002]. Although type safety with regard to programming languages in general is not easy to define, the common understanding of the property is reflected in the definition given by V. Saraswat:

- “A language is type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data.” [Saraswat 1997]

In order to ensure type safety in a system of distributed, cooperating entities, one can write them in the same programming language, compile them at the same time and then distribute to the execution environments. Such approach is taken in development of closed systems, which are not expected to be integrated with external entities. The need of systems integration lead to the development of methods of externalizing meta-data describing the type system used in particular distributed system.

Languages such as CORBA Interface Definition Language (IDL)[OMG 2004] and Web Service Definition Language (WSDL)[Christensen 2004] represent not only the interfaces of communicating remote objects, but also provide means for defining the type systems and types of exchanged messages. As a more data-centric example, consider XML Schema[Thompson 2004], which is a language dedicated solely to define the rules that bound the syntax of documents, or Resource Description Framework (RDF)[Klyne 2004].

Externalization of meta-data provides for integrating distributed system components written in different programming languages. However, in order to benefit from that, standards and mechanisms for mapping the meta-data to the implementation languages are necessary. Typically, the meta specifications are processed by tools dedicated to a particular programming environment. As a result, the tools produce sets of programming language constructs used to create, modify and destroy data of types defined in the meta specification, as well as to check for compliance with particular type at runtime.

2.2.2. Message structure

The syntax of exchanged messages is crucial to the message processing systems. Therefore, the term “data type” with regard to exchanged messages may be understood as “structural metadata associated with digital data that indicates the digital format or the application used to process the data” [Arms 1999].

Structured messages (or messages represented as documents structured in accordance with the structural metadata) are a common concept found in many middleware platforms. As a good example, consider CORBA Event Service or CORBA Notification Service[OMG 2004] that are asynchronous (message oriented) services implemented atop synchronous (method invocation oriented) middleware. In order to enable user-defined filtration of messages, the messages are structured as depicted in Fig. 10.

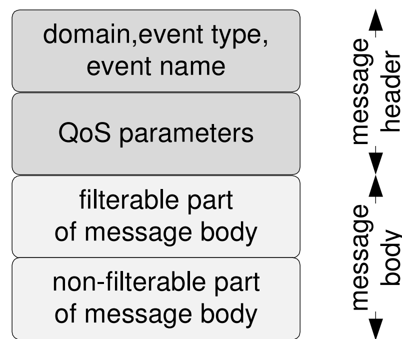


Fig. 10. The structure of CORBA Notification Service message.

The structure seems generic enough to be adopted to almost any message-oriented middleware. It consists of metadata regarding message addressing, metadata regarding system functionality (QoS), data for user-defined filtration mechanisms (which can either contain metadata or not) and a part that is inaccessible for the middleware.

A main question to be answered when deciding upon message structure is whether the middleware-inaccessible part should be kept or not. On one hand, keeping a part of each message private for the user applications seems desirable in terms of providing support for data confidentiality. On the other, that stands in contrast with the key idea of the proposed framework, which is to allow a user application to delegate message processing to the peer-to-peer environment. Moreover, putting a "visible/invisible" boundary inside the message does not make it unreadable for sniffers etc. It seems more practical to leave the decision of what parts are processed by the filters to the user applications.

It is not necessary that the source of messages should be the source of all metadata carried within them. For example, messages sent over a network in Jini[Waldo 2003] environment consist of four fields: reference of the message source, event identifier, sequence number and a "handback" object. The handback object is intended to carry data created by the registering entity and sent to the message source at registration time. The message source replicates the object along with each produced message. The meta-information carried inside the handback object can contain e.g. filtration rules to be executed on a powerful machine in order to decide whether the message is relevant to the target receiver (the target receiver's address may also be carried inside the "handback").

A significant drawback of the Jini approach is that the handback objects cannot change, so they are capable of representing only static data. Therefore it is not clear whether to adopt the handback concept directly or not. Although it seems that ability to tag the messages with metadata produced by entities other than the message sources may be desirable for framework users, the same functionality can be achieved by using message processors only for tagging the messages with pre-configured data.

2.2.3. Representation of metadata related to document syntax

In order to make the produced messages usable for the consumers, a message producer needs to publish the metadata regarding the type of the messages. Therefore, a type externalization method (and language) should be chosen. Because the published messages are expected to undergo multiple stages of processing, the externalization method should provide for easy integration with processing software. There are many approaches to solve the problem, that could be roughly classified as either API-centric or data-centric. This section presents two examples of technologies belonging to the two classes, respectively.

<u>Document Object Model</u>

The Document Object Model (DOM) is an application programmers interface (API) that was invented to allow programs and scripts to dynamically access and update the content, style and structure of documents [DOM 1998]. The models that represent documents are built either from existing data or by using appropriate programming languages constructs. In DOM, documents are represented as sets of trees that reflect the internal structure of the document. Because a model created in accordance with DOM represents only the syntax of a document, the document's semantics must be embedded in the processing code.

The DOM does not define any abstract language that would describe the structure of documents. Because of that, there is no standard way of publishing the models apart from actual documents. The only possibility to share the model between machines is to create a "prototype" document that would represent the full DOM tree. Note that this does not solve the problems related e.g. to boundary values of particular fields.

Because there is no abstract representation of document structure, queries related to the documents must be implemented as programming language constructs. Therefore, choosing DOM as the base for any distributed message processing application makes its components hard to extend or reuse.

<u>XML Schema, XPath, XQuery</u>

XML Schema, as opposed to DOM, is a standardized XML document structure definition language, but does not define any standard API to manipulate the contents of the documents themselves. Nonetheless, programming packages that provide document creation/modification API's exist. Typically, they API's are generated by tools that process schema definition documents to create programming language constructs that reflect the schema (JAXB[Ort 2003], XMLBeans[McLaughlin 2006]). The programming packages are mature enough to support important features, as checking the schema compliance with regard to documents, document introspection, etc. Note that the schema definition documents are also written as XML documents, so they also can be processed automatically.

XML Schema is an expressive language that allows defining complex documents, containing fields of many types, etc. Mechanisms such as subtyping, restricting the set of valid values, assigning default values, etc. are also provided. However, the language does not define any mechanisms for extracting parts of (possibly very large and complex) documents. Another two standard languages are used for that purpose: XPath[Clark 1999] and XQuery[Boag 2007].

XML documents have a hierarchical structure. XPath allows its user to specify the path in the document tree from its root to the node of interest. The target node does not have to be a leaf – it may be a root of a subtree of interest. By using XPath one can implement simple filtering of documents, just as in case of COM+ environment, which contains an asynchronous messaging service similar to the CORBA Event Service.

COM+ components can also use Microsoft's message queuing service (MSMQ), which relies on buffering messages on producer's side before sending them in one bundle to the consumer [Fischer 2005]. Moreover, the service is capable of buffering messages addressed to intermittently connected clients, so the final document received by the client can be voluminous. By using XPath-based filters, the message consumers can not only reduce the number of received messages, but also select the parts of messages they wants to receive.

Examples of services built upon XPath include also XRoute[Chand 2003] and ONYX[Diao 2004].

XQuery extends the capabilities of XPath by defining search operators. As a result of processing, XQuery interpreters may return a collection of subtrees that match the query specified by the issuer. The issuer no longer needs to know the exact path to the interesting part of structured document, so the processing is not as tightly coupled to the document syntax as in the case of XPath.

XML Schema accompanied with XPath or XQuery is a possible choice for syntax-based message passing/processing systems. Although syntax-based, they support creating flexible and expressive environments.

2.2.4. Representation of metadata related to document semantics

The knowledge of message semantics is crucial for ensuring proper results of message processing. In statically bound distributed systems, the system developer is required to possess the knowledge at the system deployment time. In dynamic deployment (and binding) scenario, the task is performed by the middleware. Therefore, just as in the case of syntax-related metadata, the semantic metadata needs to be externalized and published. That results in the need for choosing a technology to implement the externalization. This subsection overviews standard semantic metadata representation and query languages.

Resource Description Framework

As stated in [Klyne 2004], one of the main goals the Resource Description Framework (RDF) was created for is to provide support for automatic processing of information available in the World Wide Web. The RDF defines a language that is used to externalize semantic metadata. RDF can be used in proprietary (isolated) distributed systems, but the design goal of the creators was to support open information models used e.g. by software agents processing information available on the Web.

The RDF presents the metadata layer as a graph of labeled triples, where subjects and objects of described relationships are represented as nodes connected by edges that represent known predicates. In order to distinguish predicates that may have the same names but different meanings, RDF uses the concept of XML namespaces[Bray 2006]. RDF also leverages XML syntax[Beckett 2004] and XML Schema data types [Biron 2004], introducing only minimal enhancements. Note that the RDF-based metadata layer is extensible, i.e. it allows introducing new data types. A number of application programmer interfaces that allow manipulating RDF constructs programmatically exist[Powers 2003]. Although it is technically possible to bind RDF documents to applications with toolkits dedicated to XML binding, it is more reasonable to use dedicated ones, such as RAP [RAP] or Jena RDF API[Deng 2001][Verzulli 2001].

A number of query languages has been defined for the RDF graphs. The languages differ mainly in terms of their expressiveness [Haase 2005], but also in terms of closure, adequacy, orthogonality and safety [Haase 2004]. From the proposed framework point of view, orthogonality and safety are of particular importance.

A query language is referred to as orthogonal if the syntax of queries do not depend on the context they are used in. Note that the term “context” is very broad and can include e.g. processing node placement, which should be unknown to the query issuer (message

consumer). If the chosen language is not orthogonal, particular attention should be put to the contents of the query context.

A language is referred to as safe only if all syntactically correct queries return finite sets of results. An unsafe language may cause a processing node to enter an infinite loop. In case of using unsafe query language, either multi-thread processing nodes must be considered or only a subset of the language must be allowed.

If RDF is chosen as a way to externalize semantic metadata, SPARQL – being standardized by W3C in 2008 [Prud'hommeaux 2008] – is a natural candidate for the accompanying query language. Simple SPARQL queries resemble the commonly known SQL "SELECT…WHERE…ORDER BY" construct. The results of the queries may be simple (single fields) or complex (sets of RDF graphs). SPARQL also supports including optional parts of patterns for returned results, i.e. specifying what information is interesting, but not crucial for the query issuer. Note that SPARQL is purely a query language, it does not define any equivalent for SQL's INSERT, UPDATE or DELETE statements. These functions are handled by API's of RDF storage systems.

<u>Web Ontology Language (OWL)</u>

OWL[Patel-Schneider 2004] is a language for defining and instantiating ontologies i.e. for externalizing semantic metadata. It extends RDF by providing vocabulary enhancements for describing properties and classes and assigning formal semantics to some of the constructs. It introduces e.g. richer typing (compared to RDF), relations between classes and cardinality related constructs.

OWL defines three sublanguages[McGuinness 2004]:

- OWL Lite, designed to support classification hierarchy and simple constraints,
- OWL-DL, which offers expressiveness limited by the requirement of computational completeness, i.e. contains all OWL constructs, but puts restrictions on their usage,
- OWL Full, which offers maximum expressiveness, but does not guarantee that all queries are computable.

OWL Lite is a subset of OWL-DL, and OWL-DL is a subset of OWL Full. In general, an RDF document can be processed as an OWL Full document. Any OWL Lite, DL or Full document can be processed as an RDF document. Similarly to RDF case, OWL documents can be manipulated using dedicated APIs.

The syntax of an OWL ontology contains a sequence of annotations, axioms and facts. Annotations are used to include information related to the ontology, e.g. import statements referring to other ontologies. Axioms are used to associate partial or complete characteristics of classes to the classes themselves. Facts associate individuals with classes and state whether two individuals are distinct or not.

There is no standard query language dedicated for OWL (although candidates, like the OWL-QL[Fikes 2003] language, were created). However, because any OWL document is a valid RDF document, SPARQL can be used to query OWL documents as well.

2.2.5. Choices for the proposed framework

While the syntax-related metadata allows a consumer to process a message, the semantic metadata may provide for choosing the right source of messages. In order to support such both discovery and processing, message producers need to externalize and publish both the syntactic and semantic metadata. The semantic metadata needs to include

the producer as an individual associated with published messages by a “publishes” or “published by” relation or as a “publisher ID” property.

Although many technologies support externalization of syntactic or semantic metadata, using standard ones (i.e. XML Schema, RDF, OWL, SPARQL) in the framework will make design and implementations of its instances easier. However, because many research efforts related to semantic metadata are conducted, it may be worthwhile to design the framework so that it could be easily extended to process non-standard message semantics description languages.

2.3. Runtime environment virtualization

Tannenbaum [Tannenbaum 2002] gives MOM an analogy of postal service, in which the messages are delivered by the sender to the post office; the postal service is responsible for delivering them to the receivers. However, the analogy is good only to illustrate the processing sequences of senders, message oriented middleware (MOM) and receivers. It does not explain well what the message itself is. First, the letters (i.e. messages produced by a real-life sender) are addressed to a selected receiver, while the messages published via a MOM may not have any destination address. Second, the real-life letters are sent in a number of copies known to the sender, while in case of one-to-many MOM the number may vary.

The destinations as well as routes of messages are determined by addresses. However, the addresses are interpreted in context of meta-information stored in the messaging system. As an example, consider content-based addressing scheme, in which the message content itself does determine the message target only when matched against queries submitted by message consumers.

Introduction of specific addressing schemes is a part of virtualization of complex, real-life networks. In order to make the development of distributed applications easier, mechanisms such as address translation, tunneling, firewall traversal are hidden from the developer by middleware by a unified, simple to understand addressing. This section overviews two basic building blocks of overlay networks, i.e. introduction of virtual topology and logical messaging channels (2.3.1) and then discusses various options for logical addressing used in overlay networks (2.3.2). The section ends with a summary (2.3.3).

2.3.1. Overlay networks

Overlay networks virtualize the real (underlay) networks to provide their users with abstractions, easy to understand and use. The term “overlay network” may be defined as follows:

“An overlay network is a virtual network of nodes and logical links that is built on top of an existing network with the purpose to implement a network service that is not available in the existing network.” [Stoica 2003]

As stated by the definition, contemporary overlay networks are based on two most important abstractions: virtual network of nodes and logical (virtual) channels.

<u>Virtual topology</u>

Although the definition does neither require nor suggest that, most of research efforts in the overlay networking middleware are concentrated upon providing new services

upon the Internet. From the traditional OSI model's point of view, the networks are using a bunch of protocols situated at four topmost layers of the protocol stack.

Fig. 11 illustrates the idea of implementing logical (virtual) overlay network topology upon an existing network.

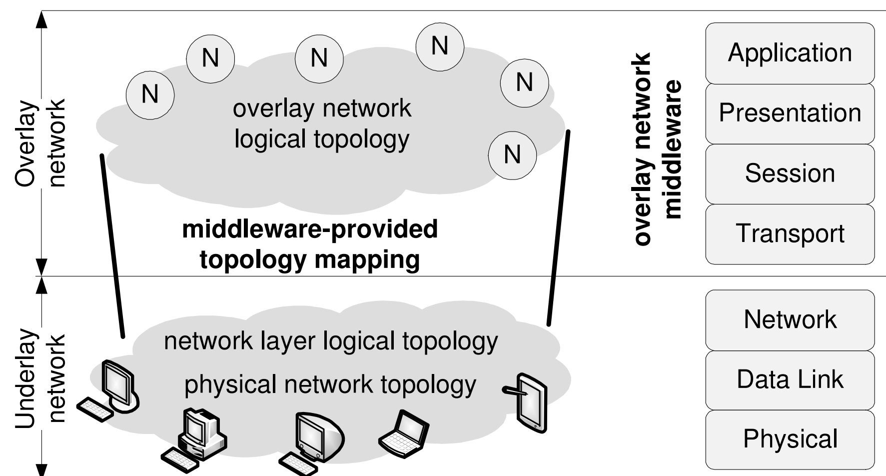


Fig. 11. The idea of middleware-implemented overlay networks.

The basic building block of overlay network topology is a node (N). A single host may be shared by a number of nodes. Even if the nodes share a host, they remain separated in the logical topology, so the communication between processes executed on the nodes is performed in the same way. Each node is capable of executing one or more network services or user applications.

Virtual communication channels

The message passing systems provide a high level of abstraction to their users. Operating on application layer, they hide the complexity of lower layers and present the user with convenient abstractions. Among them, application layer communication channels are of particular interest. A primary advantage of the channels is their ability use various transport protocols for encapsulating the routed message on its path to the destination. The transport protocols are not always OSI transport layer protocols[Flenner 2003]. From the application layer network point of view, any protocol which can be used to encapsulate the overlay network compatible message, is referred to as transport protocol. Fig. 12 illustrates the idea of logical communication channel using multiple transport connections.

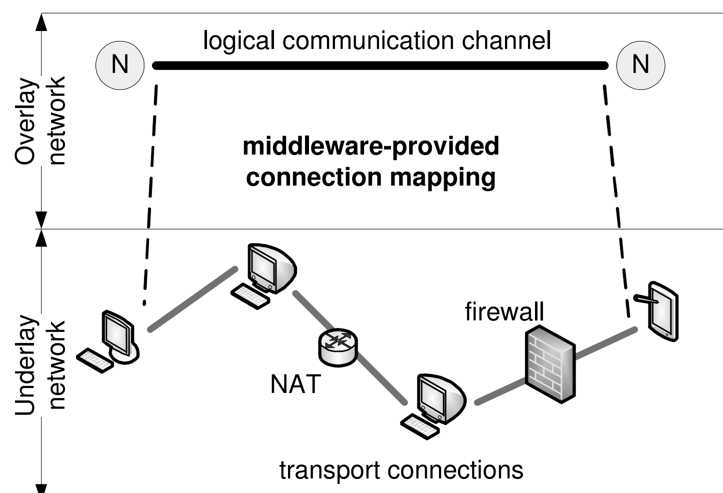


Fig. 12. Mapping between a logical communication channel and multiple transport connections.

As a major benefit from introducing logical message channels, alleviation of the communication obstacles of Internet, e.g. NAT/PAT and firewalls should be considered. Because of the implementation and runtime environment complexity, the virtual communication channels rarely offer sophisticated quality of service mechanisms. In many cases, their users may choose only between reliable or best-effort delivery.

2.3.2. Addressing and communication patterns

Most of computer networks require a way of identifying their members – in this aspect peer-to-peer overlay networks are not an exception. Over time, as the range applications of computer networks grew and so was their diversity, the addressing schemes reflected that process.

This section briefly overviews the different kinds of addressing and communication in general, stressing the features that peer-to-peer message processing systems might benefit from. It starts with regard to the set of members identified by particular kinds of addresses, and then discusses what information is carried inside the addresses.

<u>Unicast</u>

The unicast addressing scheme is the most straightforward one. Briefly speaking, each member of a network is assigned an address (typically a number) that is unique in the scope of the network. The address is then used in the messages targeted for that member as the destination address. If a message is addressed correctly, it is the task of the network to deliver the message to the specified recipient. Depending on the network, other members either do not receive the message or disregard it. Commonly used networks, e.g. Ethernet/IEEE 802.3 are based on unicast machine addressing. Note that not all networks use machine addresses. For example, widely used Frame Relay networks assign addresses to connections, not to the machines.

Overlay networks created by middleware environments also make extensive use of unicast. Because the networks are created by processes or components rather than hosts, the addresses identify the processes, not the machines. The middleware is responsible not only for finding the machine, the target is executed on, but also for delivering the message to the receiver. As examples of such addresses, consider JXTA peer identifier [JXTA 2004] (identifies a basic JXTA overlay network element) or CORBA's Interoperable Object Reference (IOR) [IONA 2000] (identifies an object). Note that the process or object identifiers can either carry information about the host of residence (as in the case of IOR), or rely on address resolution mechanisms (as in the case of JXTA peer identifier).

An important issue, with regard to middleware, is the address persistency. Some of the middleware environments provide unicast address persistency e.g. for dynamically activated entities. For example, in Java RMI[Sun 1999], when a long unused object is deactivated and activated again, its address does not change (provided that the object was registered with the middleware as an activatable object) [Sommers 2000]. Another example is CORBA's Portable Object Adapter (POA), which also provides address persistency [Gray 2004].

<u>Multicast</u>

In order to facilitate group communication, multicast addressing scheme is introduced by many networks or middleware environments. In multicast capable networks, entities join the multicast groups and leave them dynamically. Messages sent to a group

address are delivered to each member of the target group. Whether the delivery is guaranteed, or not, depends on the protocols in use. Group communication based on multicast can save much bandwidth when compared to simple unicast communication between the message sender and each of the receivers. However, IP level multicast is rarely supported in wide area networks, so in order to provide the group communication ability to the users one need either to provide IP level tunnels between so-called “multicast islands”, or rely on application level multicast.

In order to provide multicast functionality at application level, middleware technologies leverage ideas from network layer, such as the algorithms for joining/leaving multicast groups, optimizing multicast trees and ensuring message delivery. In some cases they use the network layer transport protocols directly (when operating inside multicast islands) [JXTA-RM 2005], while others (like Scribe [Castro 2002]) implement the mechanisms on their own [Castro 2003-1]. Both approaches have their advantages and disadvantages. The network layer based approach requires configuring routers to support tunneling between multicast islands, but offers better efficiency. Application layer based approach sacrifices efficiency for flexibility and simplicity offered by uniform overlay topology. An interesting research effort was presented by X. Jin et al. who constructed hybrid environment for supporting multimedia streaming. The multicast trees of Scalable Island Multicast protocol (SIM) [Jin 2006], which forms a basis for the streaming environment, are built at application level, but they include automatically detected network-level multicast islands. The performance studies conducted by the designers of the protocol [Jin 2007] show that the hybrid approach offers both considerably lower end-to-end delay and resource usage compared to Narada [Chu 2002] and Overcast [Janotti 2000] while offering the same functionality.

Broadcast

Broadcast addresses are well-known addresses that refer to “every member of the network”. Commonly used in local area networks e.g. for network organization and management purposes by protocols like ARP[Plummer 1982], they are feared of in IP networks, as their abuse may result in network breakdowns. Because of the diversity of machines connected even to a single local area network, they add little value in comparison with multicast. Therefore, version 6 of Internet Protocol (IPv6) does not define any broadcast addresses [Hinden 2006]. Even the hardware address resolution, in IPv6 protocol stack implemented as one of neighbor discovery mechanisms[Narten 2007], is based on multicast instead.

Overlay networks created by middleware environments make little (if any) use of broadcasts. If really needed, organizing multicast groups each member of the network is required to participate, serves as well.

Anycast

Load balancing techniques are widely used to scale services to the growing demand. Anycast addressing is one of concepts used in load balancing. The concept of network-layer anycast was introduced in RFC 1546 [Partridge 1993]. According to the idea, network members that provide the same functionality may share a single anycast address, forming an anycast group. A message sent to an anycast address is delivered to any of the group members. The selection of the recipient is performed by the network devices.

Anycast addresses may be syntactically identical to unicast addresses, as in the case of IPv6[Hashimoto 2006]. In case of IPv6, the packets sent to an anycast address are

delivered to the “nearest” (according to the routing protocol metrics) possible recipient. Currently, IPv6 anycast addresses are assigned only to routers.

Pure network-layer load balancing among anycast group members is feasible as long as there is no session created between communicating parties (i.e. the exchange between the parties follows the simple request/reply pattern). In other case, support from higher layers is required. Moreover, network layer metrics, although useful, do not provide enough data to perform the load balancing efficiently. As an example of a “missing” metric, consider the load of anycast group members. Therefore, in order to perform efficient load balancing between a group of servers, application layer anycasting, which may be based on broader spectrum of metrics, may be considered [Bhattacharjee 2002]. Application-layer anycast is supported e.g. in Scribe, a topic-based publish/subscribe environment built upon Pastry [Castro 2003-2].

Contents of addresses

Any kind of address must be resolved to a set of entities (network members) identified by the address. The resolution is performed by the network, transparently from the message sender point of view. The address resolution mechanisms interpret the data specified by the sender as a destination address in context of knowledge (either stored explicitly in a database, or determined dynamically) describing the network environment. Depending on the approach the network constructors taken, the contextual information can help in resolving different kinds of addresses, which can be used by user applications. This section describes briefly three kinds of addresses, which seem to be most interesting from an advanced publish/subscribe system point of view.

References

A reference is a numeric or stringified identifier of the addressed object. Depending on the scenario, the reference can have host-local (e.g. high level programming language object), network-local (e.g. an identifier of a JXTA peer in a custom peer group), or global (e.g. an IP address) scope of usability. Addresses of this kind can refer to one or more entities.

Reference-based addresses are added to the contents as meta-information created and interpreted by middleware environment. Depending on the platform, the addresses may be organized either in a flat or hierarchical space.

Reference-based addresses are typically assigned either to machines or – especially in dynamic environments – to resources. Usage of resource addressing is even considered one of the key characteristics peer-to-peer content-oriented networks [Shirky 2000].

Topic-based addressing

A topic corresponds to a logical channel connecting the producers and consumers of messages. In an ideal case, the channel links each of the producers to all of the consumers [Baldoni 2005]. With the use of topic-based addressing, both the producers and consumers in their communication refer to the logical channel address rather than directly to themselves. The channels decouple the producers and consumers and make them anonymous.

In order to enable the middleware to discriminate between messages published to different topics, the names of the topics (which are typically human-readable character strings) are carried as meta-information within the messages. The names of the topics act as logical destination addresses, that are resolved by messaging middleware. Contrary to the

references, topics are created by application developers and therefore may include a little semantic information that may e.g. improve the readability of introspection results to the end user. However, the information is of no use to the middleware.

The limited expressiveness of topics is a bit improved by introducing hierarchical organization of topics [Baehni 2004]. In such case, a topic can have multiple subtopics. Each message directed to a subtopic is also directed to the enclosing one. Therefore, consumers connected to the enclosing topics receive messages from the subtopics as well, while the consumers connected to a subtopic do not receive messages from the enclosing ones.

There are a few terms that refer to the topic-based addressing. Baldoni and Virgilito [Baldoni 2005] postulate to refer to hierarchically organized topics as subjects. On the other hand, CORBA Event Service [OMG 2000-1] introduces a name of channel-based to the topic-based notifications if the topic names are not included in the messages. Nonetheless, the principles of message delivery remain the same for all the cases.

Content-based addressing

At a first glance, the main difference between content-based and other kinds of addresses it that the content-based addresses are not carried as a meta-information. Rather, the meta-information is a result of an analysis of messages' internals. The message contents are used in message routing process to extract the information required to identify the recipient(s). That makes the message contents an address, referred to as content-based address.

The analysis is conducted in a way defined by a particular middleware environment and in accordance with the criteria specified by message consumers. There are multiple middleware platforms that rely on processing of the messages contents. Their abilities are commonly limited by languages used for marshalling messages and expressing queries. First large-scale content-based message distribution systems used simple documents built from name-value pairs. The usage of XML introduced the possibility of using hierarchically structured documents for that purpose.

Documents used to represent the messages can be processed by the middleware using expressions of query languages, either platform-dependent or standardized. However, decoupling message publishers and subscribers by messaging services does not make them completely independent. For proper interpretation of the message contents, they need to share knowledge about message semantics. The knowledge regarding message semantics may be distributed, i.e. hard-coded into the end user applications. In such case, the entities responsible for routing and processing of messages are limited to operate on message syntax level.

In order to standardize and structure the knowledge, a system developer may introduce semantic database, known to both sides of communication. Modern systems based on the publish/subscribe paradigm may refer to the database also when deciding where to send a message[Keeney 2008]. The case of semantically expressed content-based addressing scheme is known as "concept-based addressing". When using concept-based addressing, semantic descriptions of entities, either kept in a single database or not, are used to direct messages. Concept-based addressing may discriminate between common and domain specific ontologies [MIX] or use just one scope.

2.3.3. Choices for the message processing overlay networks

Because the message processing overlays are going to be created on user application's demand, putting heavy constraints on their topology could severely limit their usability. Therefore, unstructured topology seems to be a safest (though not in all cases most efficient) choice.

Message channels are the basic means of underlay network virtualization. Because the underlay networks are very complex, implementation of logical message channels is one of the key user requirements regarding any message passing and processing system. Therefore, the framework must implement the abstraction or build upon an existing solution.

There are at least two addressing schemes needed by message processing overlay networks: addressing of message sources and addressing of network internal entities, i.e. message processing nodes. The primary requirement for addresses of message sources is to support searching process and allow for expressive queries. On the other hand, the primary requirement for addresses of message processing nodes is to support efficient message passing between the nodes.

Among the surveyed kinds of addresses, concept-based addresses seem to be the most expressive, and therefore most suitable for using with regard to the message sources, although the number of identified nodes is not determined a priori. Therefore, it would be safe to assume that concept-based addresses identify a group of possible equivalent message sources. In order to connect the sources to a messaging channel, additional lower-level mechanisms that map concept-based addresses to sets of unicast references, should be used. Applying dual addressing scheme with regard to the sources also enables the users of the framework to change the upper layer scheme in case it does not suit their needs. Moreover, from the framework designer's point of view, it is also convenient not to bind the framework to a single addressing scheme and make it more customizable.

Simple message passing that occurs between message processing nodes does not seem to need more than simple references. Therefore, the addresses defined by any underlying technology should provide enough expressiveness. Nonetheless, in that case, introducing private per-network addressing scheme could result in easier portability of the whole framework. If the address mapping functionality is designed as a separate software module, change of underlying technologies would be much easier. Note that the addresses of the message processing nodes should be unique not only inside the network, but inside a container as well. As a solution, assigning addresses to whole overlay networks should be considered. When such solution is chosen, a tuple (network address, node address) can be used as a system-unique message processing node address.

Before the message passing takes place, a message processing system must interpret a consumer's query, find message sources, create message processing nodes, deploy them etc. These functions should be performed by system internal entities. Taking into consideration that the system is expected to run in changing environment, at least some of the entities will need to be duplicated. Anycast addressing can help with submitting a job to any of equivalent entities. Provided that the communication channels are reusable, i.e. their ends can be substituted without destroying the channel, assigning persistent addresses to the channels (not only to the framework instance nodes) can be a way of implementing well-known anycast addresses.

Table 1 summarizes the addressing choices.

Table 1. Types of addresses to be used by framework instances.

addressed entity	address types
addressing schemes to be used by external entities	
query target	concept, topic, or reference-based anycast
query source	reference-based unicast
contract offer target	reference-based anycast
contract offer source	reference-based unicast
addressing schemes for system-internal entities	
deployed processing node	reference-based unicast
redundant internal entities	reference-based anycast
not redundant internal entities	reference-based unicast
message target	reference-based unicast or multicast
message source	reference-based unicast

The main enhancements to existing message passing and processing systems are:

- allowing multiple addressing schemes including concept-based addresses to be used by query issuers,
- use of anycasts in both system external and internal addressing.

2.4. Adaptability of framework instances

Middleware run in changing environments needs to be equipped with means for tracing the changes in order to react accordingly. Similarly, a part of applications built upon the middleware need to be informed about its state in order to adapt their operations. This section overviews key mechanisms that support the interaction between underlays, overlay-oriented middleware and applications.

2.4.1. Introspection

The growing popularity of new application development concepts, such as ubiquitous computing introduces new challenges to the middleware. Contemporary middleware platforms include mechanisms for cooperating with applications. Providing introspection mechanisms is a basic requirement to implement the cooperation.

The introspection concept comes from high-level programming languages, and in that area can be defined as “the ability of an object to reveal information about itself as an object—such as its class and superclass, the messages it can respond to, and the protocols it conforms to” [ObjC 2008]. Through the use of introspection, applications can reason upon programming language constructs. Similarly, with regard to distributed applications, introspection may e.g. help an external entity to discover the components of an application and to examine their properties, including context information, regarding e.g. the machine a particular application is run on [LeSommer 2006].

Note that the examination of distributed system structure is not enough to provide effective middleware management. In order to make management decisions, one needs to

know not only the configuration, but current state of middleware components as well. To facilitate this requirement, middleware environments come with monitoring interfaces.

In order to facilitate effective overlay network management, the framework needs to provide introspection and monitoring services. Moreover, to support adaptation, an mechanism for executing runtime measurements of key framework instance characteristics should be provided. Information gathered by monitoring subsystem may be used not only by external entities, but by framework instance internals as well. Both adaptation (external) and reflection (internal) mechanisms need monitoring data to perform optimization actions. Therefore, not only containers but all framework instance internal entities should be accessible through the monitoring service.

2.4.2. Adaptation

A middleware platform is considered adaptive if its functional behavior can be dynamically modified in response to a change of requirements of environmental conditions [Loyall 2001]. The modification is initiated by an external entity, e.g. middleware user, application or administrator. Because the changes are expected to occur at runtime, particular stress should be put on authorization of change initiators and the bounds of adaptation, expressed e.g. in adaptation policies.

In case of the proposed middleware, adaptation should be considered with regard to optimization of overlay networks processing. Optimization of overlay networks is somehow problematic, because the optimization criteria could only be roughly defined. Because the optimization strategy depends on the specifics of framework instance and respective overlay networks, the framework should support gathering monitoring data and performing actions such as adjusting control parameters of a message processing node or component migration rather than implement a specific optimization strategy.

2.4.3. Reflection

The previous subsection, which outlined introspection, was focused on providing entities external to the middleware with means to inspect or alter its behavior. In such approach, the adaptation driving mechanisms are placed outside the middleware. Reflective middleware is constructed in a different way. Although the interfaces that allow alterations to the middleware behavior may be exposed, the middleware is capable of self-adaptation. Therefore, the reflective middleware environments add self-awareness [Kon 2002], to the mechanisms that built up adaptation mechanisms. Changes in the environment are reflected in the self-representation data and may result in modifications in middleware behavior. The causal connection along with self-representation are considered the key properties of a reflective middleware [Coulson 2002]. As examples of a reflective middleware environment, consider OpenCOM[Pissias 2008], or Hermes[Pietzuch 2004], which uses a peer-to-peer overlay network of message brokers.

High availability of key framework instance internal entities is certainly one of the most important aspects of middleware reflectiveness. Internet-based systems frequently use dedicated protocols and services to ensure that a specified functionality is constantly available either to the end users or to system components. As an example of a message passing/processing technology that uses high availability mechanisms, consider IBM WebSphere Message Broker[Credle 2007]. It seems reasonable that the REMP framework reuses or introduces concepts related to high availability.

As stated in the previous section, migration is considered one of key mechanisms for adapting to the environmental conditions. Therefore, although due to administrative constraints, the framework instance internal entities are not expected to be as mobile as user-defined components, migration of instance internals should be considered. Among other mechanisms that support self-adaptiveness, automatic instantiation and activation of message processing system internals is of particular interest. In order to be dynamically deployable or mobile, the internal entities should be implemented as self-contained components. In an ideal case, the components should be compatible with containers used for hosting message processing nodes. That would provide for e.g. keeping a minimal number of entities of particular kind running. Implementing all kinds of internal entities, including containers, as deployable components would introduce great possibilities of framework instance virtualization. Note that Java 2 compliant runtime environments are capable of loading classes dynamically. The class loading mechanism interface can also be used by application developers in order to load remote libraries on demand [Eisenbach 2002][Ławniczek 2003] and forms a base for development of component-oriented applications.

The capabilities of components are to some extent determined by the hosting container. In case of components implementing functionality of framework instance internals, it is very important that the container provides them with access to key services, e.g. discovery and monitoring service. Therefore, containers should at least discriminate between "ordinary" (user-originated) and "trusted" (internal) containers. Mechanisms such as code signing and runtime security checks should be employed for that purpose.

2.4.4. Security

The term "security" is very general and understood in various ways, depending on the context. The most commonly discussed aspects of security include: availability, confidentiality, integrity, authentication, authorization and non-repudiation. Depending on the application, some of them are more and some less important. Therefore, it is not possible to select the best mechanisms for all framework applications. However, a few general observations can be made.

First of all, as the instances of the framework will be run on a changing network, availability and integrity of framework services will be crucial for most applications. Therefore, mechanisms such as replication of service instances and synchronization of their states must be provided. Second, because the instances are expected to be run over the Internet, authentication and authorization mechanisms are also desirable. Solutions that support authorization checks for particular publish/subscribe environments exist (e.g. EventGuard[Srivatsa 2005] for SIENA).

Assuring non-repudiation could require the middleware to keep track of all messages published by all entities (not only producers). Moreover, all of the messages would have to be digitally signed and replicated in case a part of the system gets disconnected. Although possible, keeping such a large database at middleware layer seems impractical, given that only a subset of supported overlays would make use of that module.

By delegating some of the processing to the middleware, the user sacrifices end-to-end confidentiality of data. However, hop-by-hop confidentiality assurance mechanisms can be implemented or reused to satisfy most of the applications. The mechanisms include encrypted communication channels, digital signatures, etc.

2.4.5. Conclusions for the proposed framework

Providing at least basic means for reflection seems to be one of the most important requirements for the proposed middleware, due to the characteristics of its foreseen runtime environment. Internet-based peer-to-peer networks tend to be dynamic and therefore in order to maintain deployed overlay networks consistency, the middleware needs to monitor the changes. Based on the monitoring results, reflection mechanisms could e.g. reinstantiate required parts of the system, reconnect the message channels, etc.

The goal of the framework is to support a wide range of message processing systems. It is hardly possible to envision an ultimate self-adaptation strategy that would suit all of the instances. Therefore, interfaces for introspecting and adapting the system need to be exposed.

Similarly, no ultimate set of security mechanisms could be selected. Instead, interfaces for security-related extensions should be provided, limiting the framework-implemented mechanisms to the most obvious ones.

2.5. Summary

The chapter overviewed various aspects related to the proposed middleware. A review of existing message passing and processing systems shows that none of them could be extended to provide the functionality requested from the framework due to their broker-oriented architecture and tight relationship with not very expressive languages for specification of consumer needs regarding message source selection and message streams processing. Moreover, the meta-data externalized by the surveyed systems is insufficient for executing queries on message semantics. Nonetheless, some concepts introduced by the broker-oriented systems (listed in Table 2) could be reused.

Table 2. The main concepts of the reviewed technologies to be reused by REMP framework.

Section	Concept	From
2.1	graph-based processing model	Gryphon
	content-based subscriptions	SIENA
	intermediary results reuse	Gryphon
	simple hop-by-hop QoS parameters	CORBA Notification Service
2.2	message syntax and semantics externalization languages: XML Schema, RDF, OWL, SPARQL	W3C Consortium recommendations
2.3	concept-based addressing	MIX
2.4	introspection and reflection	OpenCOM
	high availability	IBM WebSphere Message Broker

First of all, the commonly used graph-based processing model seems to be a reasonable choice for REMP framework. Moreover, from the framework point of view, the message source advertisements that allow the analyzed environments to optimize their processing could also be used to as a means for simple, consumer-initiated introspection.

Survey of metadata representation and addressing options, that followed, lead to selecting a combination of addressing schemes that should provide the user with maximum flexibility with regard to queries, while allowing simple to implement machine message processing. Apparently, although contemporary middleware services use various addressing schemes, they rarely operate on more than one in parallel. REMP framework is built upon an alternative approach – the list of addressing schemes used by message producers and consumers not only allows various schemes, but also is extensible. Because of the variety an extensible address interpretation/mapping service needs to be designed.

Contemporary middleware platforms provide means for interaction between their deployed instances and their runtime environment. In order not to deal with the complexity of networks connected to the Internet, reuse of existing abstractions is suggested. Environments that provide the most needed abstractions of virtual network topologies and virtual communication channels are considered as the basis for REMP. In peer-to-peer networks, which are intended to form the execution platform for REMP instances, configurations of nodes depend on their administrators. Therefore, introspection mechanisms need to be employed to check e.g. whether a component is deployable in particular container. Note also that the runtime environment contains not only the underlying networks, but the overlays overlay networks deployed by the consumers, the requirements of which are hard to predict. To point out the most important aspects of interaction with the overlays, short reviews of quality of service parameters introduced by existing middleware and services related to introspection, reflection and security were also included in the chapter.

The discussion presented in this chapter forms a basis both for Chapter 3, which presents the implementation platform independent model of a REMP framework instance, and Chapter 4, which describes the prototype implementation of the framework's core services. Both the model description and prototype implementation are built upon abstractions and services introduced in this chapter.

3. Framework instance model

The goal of this chapter is to present a model for message processing systems built with the proposed framework. According to the definition introduced in section 1.1, a message processing system is “an instance of the message processing framework, customized, deployed and configured, used by message producers and consumers”. Survey presented in Chapter 2, that was dedicated to review existing concepts and systems related to message passing and processing, shows that a message processing system should be designed not as a single black box, but rather as a set of cooperating services. This chapter identifies the services that build up REMP (section 3.1) and shows how they are intended to cooperate to create an overlay network (section 3.2), maintain existing overlays (section 3.3) and the system itself (section 3.4). Sections 3.5 is dedicated to discuss the services’ extensibility, which is expected to be the main virtue of the framework, and main options related to customization. Section 3.6 summarizes the chapter.

3.1. Overview of the framework instance architecture

In order to identify the services that build up REMP framework instance, basic functional blocks (subsystems) of the message processing system “black box” are depicted in Fig. 13. The subsystems themselves are built from services that will be discussed in the following sections.

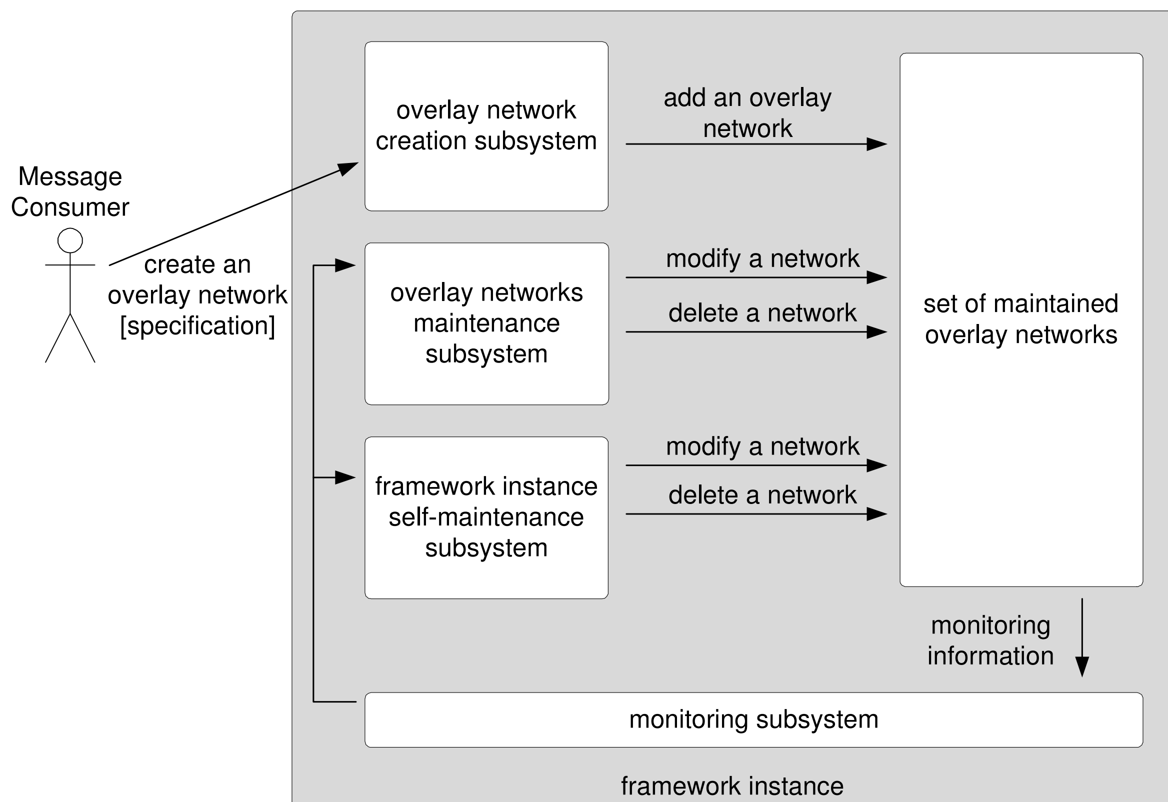


Fig. 13. The subsystems building up a REMP framework instance.

Queries submitted by message consumers, which specify the intended message sources, message streams’ processing stages and message flows are the main input for a REMP framework instance. The message processing system interprets the submitted query

as a specification of an overlay network to be created. The network created in response to a query may undergo runtime modifications, due either to failures of parts of the system infrastructure or to optimizations based on results of overlay network and the message processing system monitoring.

In other words, the instance is expected to provide its users with a shared infrastructure for creating application-specific overlay networks that implement the idea of a virtual message producer (see Fig. 2). Because the functionality of a virtual message producer is defined by a single message consumer, the overlay network defines an N-1 relationship between real message producers and a message consumer. The life cycle of the overlay network is also bound to the existence of the message consumer that specified it (Fig. 14). The network is created by message processing system internals in response to the consumer-originated query. That could be a time-consuming process, because there is no guarantee that the system will have all needed elements available instantly. Therefore, a processing timeout may occur. Then, after creation and deployment, the message processing phase takes place. During message processing (“running” state), it is monitored to detect failures and optimization possibilities. Modifications that may occur when the network is in “running” state, are expected to be transparent to the network, so the state remains unchanged. Once a message consumer disappears, the network is closed down. Note that the procedure of detecting message consumer “disappearance” may include waiting for a specified time in order not to close down networks in case of message consumers suffering from short time disconnection. The state of the network can be also turned back to “nonexistent” in case of an error the system cannot recover from.

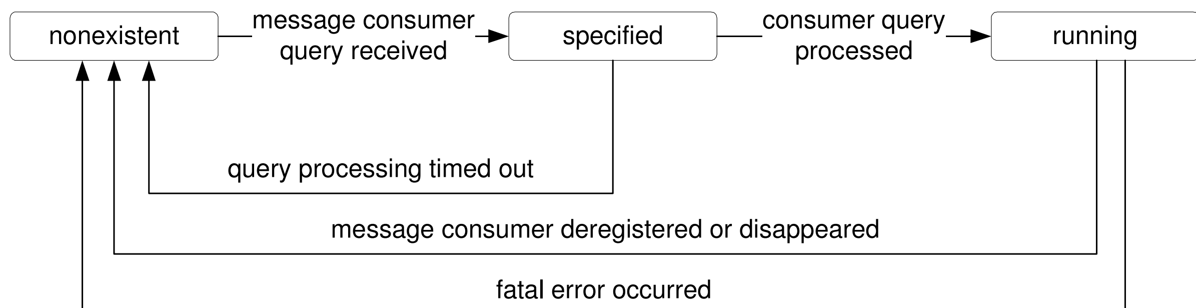


Fig. 14. The life cycle of an overlay network created by framework instances.

In order to create a message processing overlay network, components representing respective stages of processing delegated by a message consumer are deployed in selected containers that are basic building blocks of the system infrastructure. To be fully functional, the system needs not only the containers, but also some supportive elements to be configured and run. For example, the message processing source code included in the overlay network specification may need to be wrapped in generic components’ code in order to be deployable in containers. Fig. 15 illustrates elements of a configured message processing system. It is assumed that the supportive elements are configured and started prior to any overlay network deployment.

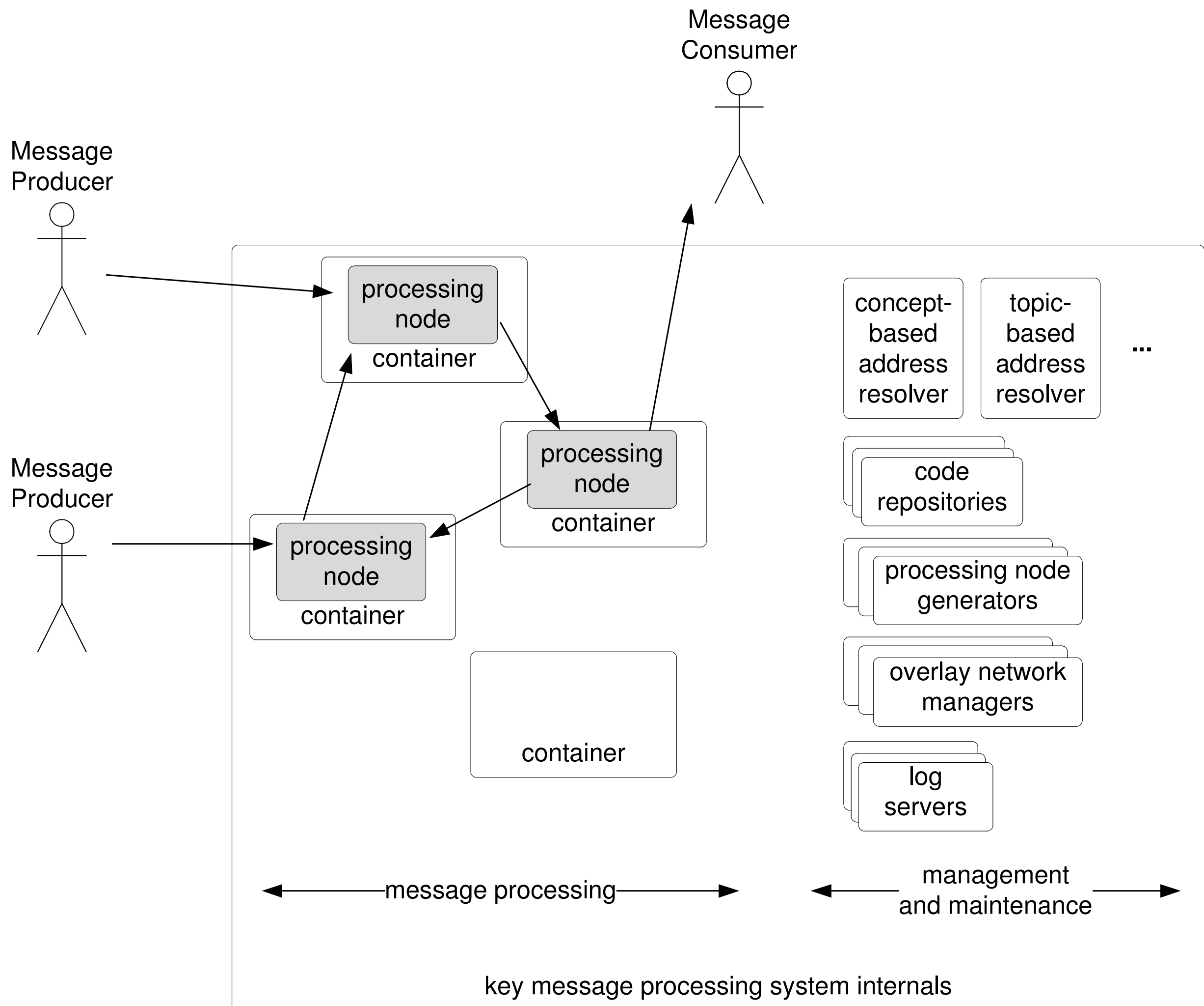


Fig. 15. Key elements of a running message processing system infrastructure.

The system infrastructure is represented as a set of services (Fig. 16) represented by application programmer's interface elements. The elements that seem to form the main customization area are shaded in gray. From the API part, the customizable blocks do not map directly to any of the services. Rather, they represent options that can be provided by framework instance designers and developers modifying the default behavior of many services during the aforementioned process of customization, deployment and configuration. The non-customizable API elements are either too simple (like the abstract message source and sink stubs), or do not expose any customization interfaces (the publish and subscribe services).

The core functionality of the framework internal services is provided by the framework itself, while the details might be changed quite easily by developers. Apart from the Binding, Discovery and Communication services, which are expected either to be provided by or tightly coupled to an underlying middleware, all other building blocks are designed with customization in mind.

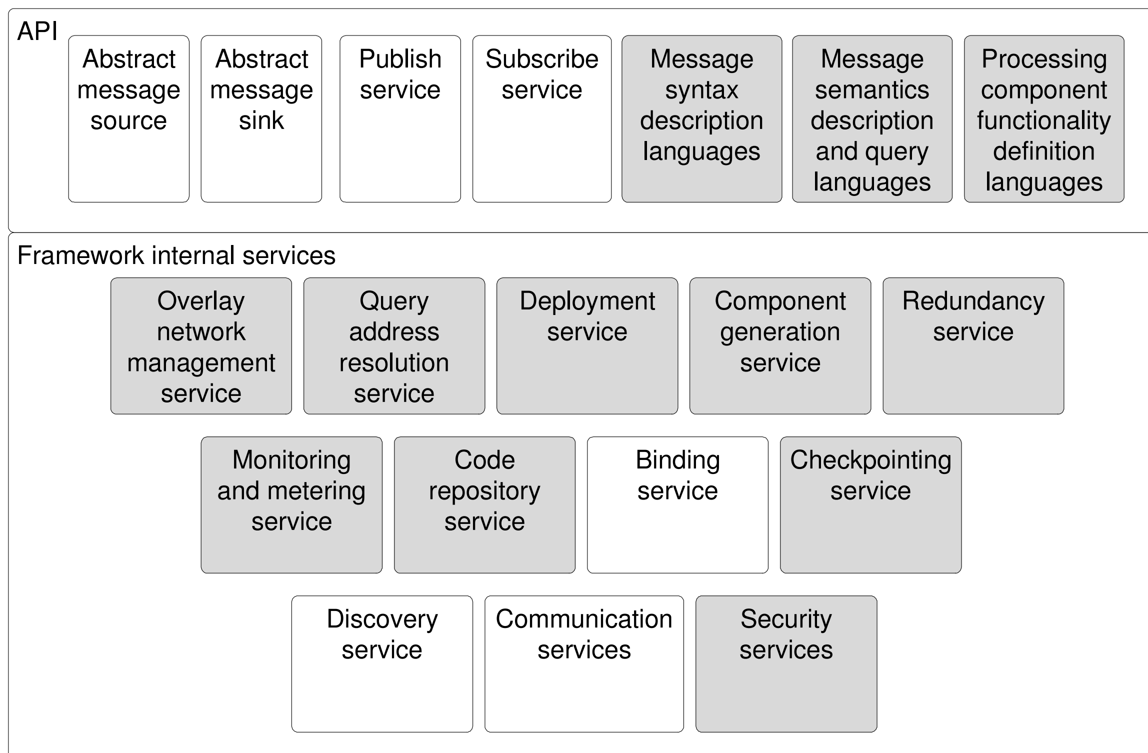


Fig. 16. Services and API elements provided by the framework.

The following sections (3.2 to 3.4) outline functionalities assigned to the framework services and describe relationships between them. Section 3.2 is dedicated to analyze the use cases related to operation of a framework instance and discuss functional services needed to create and deploy an overlay network. Sections 3.3 and 3.4 are dedicated to identify the services needed to maintain an overlay network and the whole framework, respectively. Section 3.5 focuses on key framework customization options and its extensibility.

3.2. Overlay network creation

This section is dedicated to identifying the most important services that implement the functionality requested by framework instance users. The analyzed sequences of events assume that an instance is already configured and all of its services are fully operational. A discussion of use cases (3.2.1) leads to identifying several services, which functionality is further discussed in 3.2.2.

3.2.1. Use cases

The primary functionality to be offered by any of framework instances is allowing delegation of parts of the message consumers functionality to the middleware. Therefore, creation of message processing overlay network is the main use case of any instance. From the message producers' and consumers' point of view, there are another two (much less complicated) use cases: producing messages and introspecting the framework instance to get descriptions of available streams of raw messages (Fig. 17). Other use cases (related e.g. to securing a message producer or consumer) also exist, but are related more to administration than basic functionality.

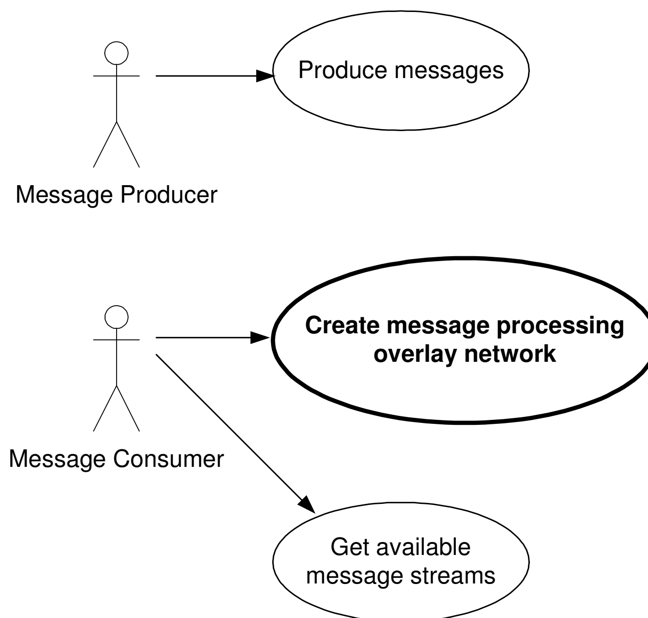


Fig. 17. The basic use cases for the system.

The following paragraphs overview each of the basic use cases.

Produce messages (Producer)

A message producer is going to provide the framework with at least one stream of messages. It is assumed that the producer will announce the availability of the message stream by issuing an advertisement. Such advertisement is expected to describe the semantics of generated messages as well as their syntax. Would it be only a simple record carrying the name of the topic the messages belong to, syntactic or semantic description of message contents, depends from the framework instance. There are at least three possibilities to propagate the advertisements inside a framework:

1. by making a message source (the object representing the message producer) register the advertisement in a pre-configured counterpart (message producers registry),
2. by making the message source discover the counterpart (registry) dynamically and then register the advertisement,
3. by providing a well known anycast address the source publishes its advertisement to.

The experience of Elvin developers shows that choosing the first approach leads to problems with scalability [Segall 2000]. Therefore, newer versions of Elvin use the second approach. However, the third method seems to allow the highest degree of flexibility, because it does not require an outside entity (a message source) to know the internals of the system and therefore does not determine the system implementation details in any way. For example, the anycast may be intercepted by a discovery agent and forwarded according to the rules specified by the instance developer (not necessarily to the first discovered registry). Moreover, the third option allows the registry to be distributed even among all the members of framework instance. Both scenarios may be implemented without changing the message sources.

Advertisements originated by message producers should contain not only the description of messages, but rather a description of contracts offered by message producer as a service. The contract should also contain e.g. quality of service parameters the source is able to support. After a corresponding query is found, the message producer may start publishing messages.

Fig. 18 illustrates the sequence of key events that occur before the message publishing takes place. First, a description of contracts offered by a message producer is registered. Since the message processing system internals are opaque to the message producer, it does not know the addressees of the advertisement. In other words, the advertisement is submitted to “an interested entity”. It may be useful to make the producer (or a message source) register its contract offers periodically by using a registration leasing scheme. Such approach would help to clean up the contents of discovery service databases by removing detached producers’ contracts.

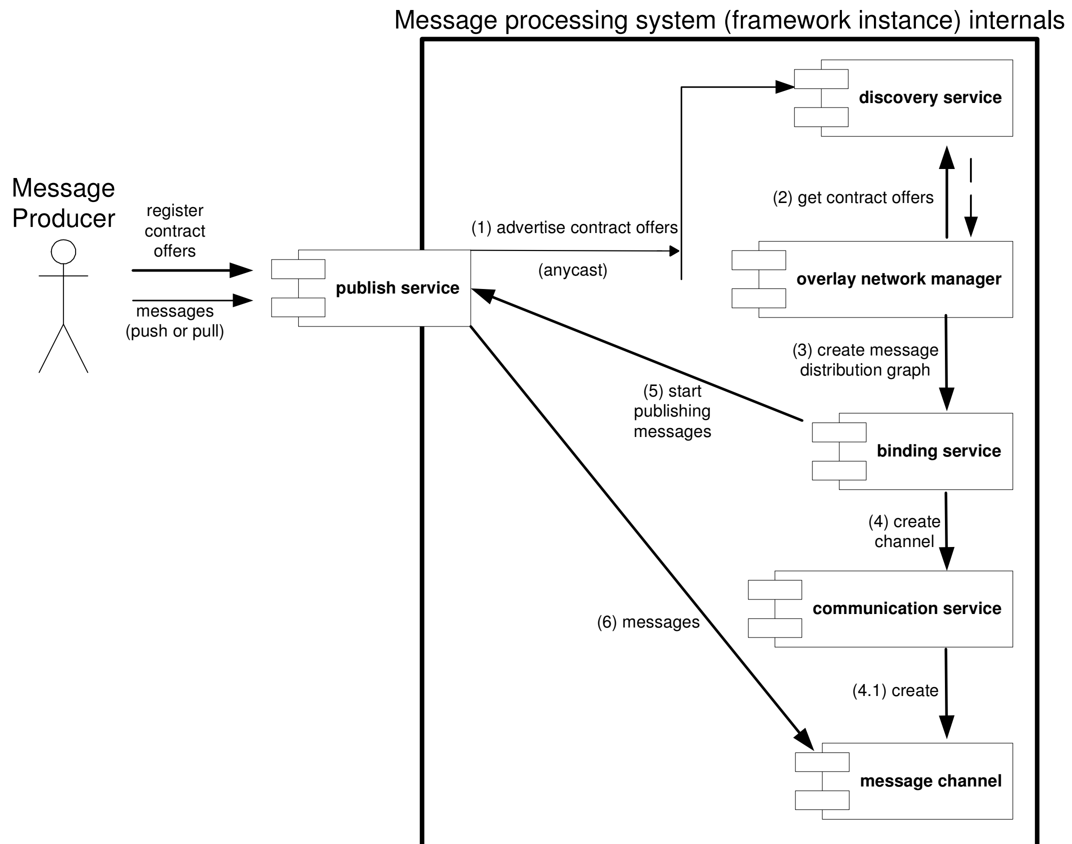


Fig. 18. A simplified collaboration diagram for the “Produce messages” use case.

The communication between the message producer and system internals is handled by the Publish Service, instance of which is expected to reside on each message producer machine. Therefore, the client application developer does not need to know the registration and communication details. By hiding the details in the Publish Service, the use of reference-based addresses (see Table 1) is also opaque to the producer client application developer.

After the contracts offers are registered, a module responsible for implementing the overlay networks (overlay network manager) may decide to use one of the contracts in a newly created message distribution graph. In such case, it will need to create a message distribution channel between the producer and another entity (either a container for message processing nodes or a message consumer). In order to create a channel, graph binding subsystem will be called, which will create (or reuse) a channel and forward its reference along with the selected contract details to the message source.

When the message producer is no longer needed, similar sequence of calls will be made. The manager of overlay network will instruct the binding subsystem to destroy the

message channel and the binding subsystem will signal the producer to stop publishing the messages.

Create message processing overlay network (Consumer)

Message consumers expect a message processing system to respond to their queries with appropriate message streams. Fig. 19 illustrates simplified process of overlay network construction.

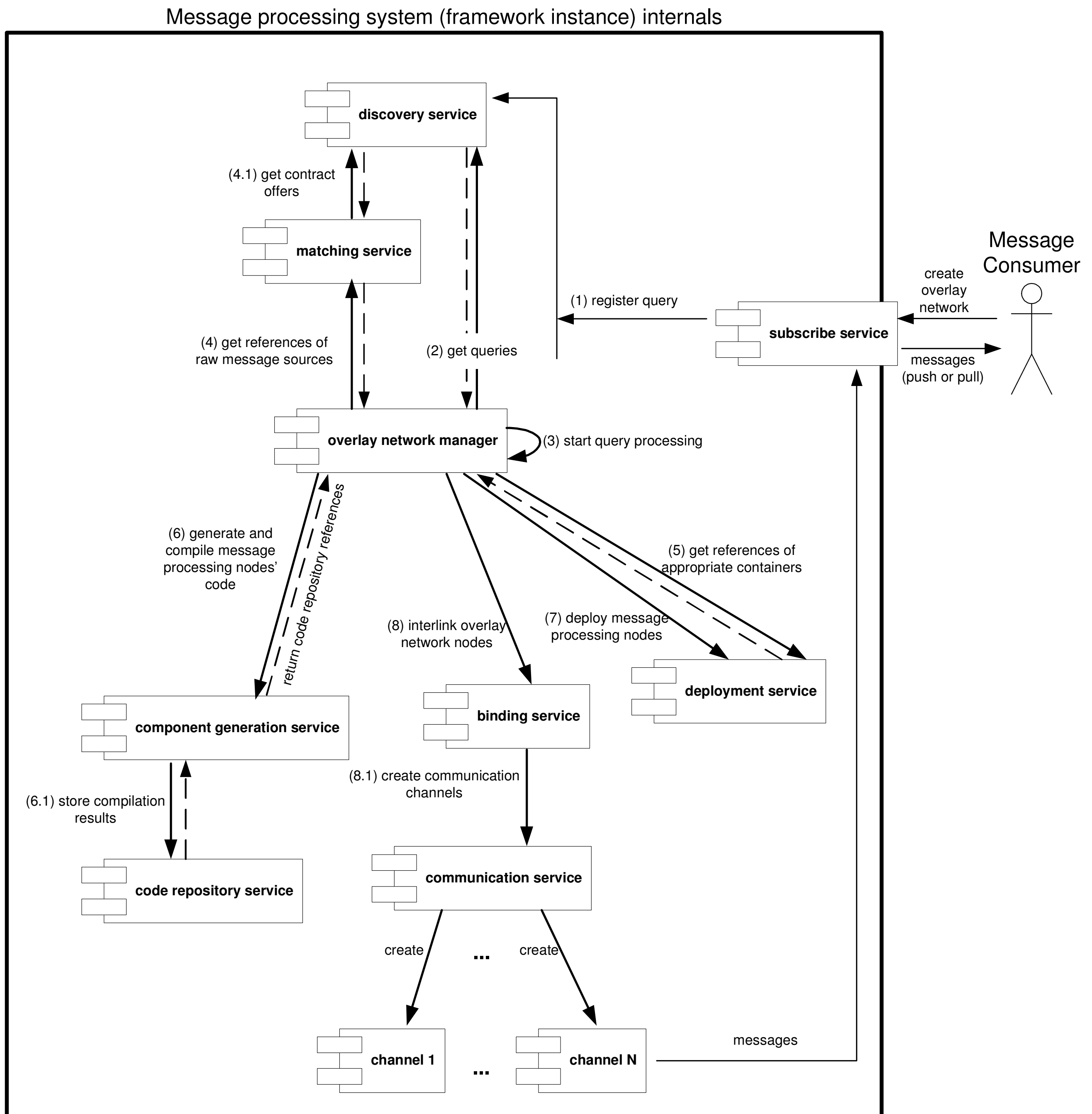


Fig. 19. The process of message processing overlay network creation (simplified).

The query is in fact a specification of an overlay network to be created by the system. Therefore, it is expected to contain:

- sub-queries regarding contract offers published by message producers,

- specifications of transformations of the messages obtained directly from the producers (called later raw messages),
- specification of message distribution graph that interlinks message producers, processing nodes and the consumer.

It is the responsibility of the framework to organize all the processing, including creation of message processing entities and communication channels. Such approach may result in reducing the system's network bandwidth requirements, especially if the processing entities would effectively reduce the volume of flowing messages by e.g. merging a few message streams close to their sources. Unless no processing is specified, the consumer will be presented with a virtual message producer that – from the consumer's point of view – will produce messages that will exactly match the specified query. In "no processing" case, the message producer will be a real one.

Just as in the case of message producers, the consumers might issue their queries either to pre-configured or discovered query handlers or published to anycast address. Similarly to the case of message producers, the anycast based approach seems to be more flexible than the others.

For the same reasons that refer to message producers, it may be beneficial not to expose system internals to message consumers. Therefore, when a message consumer issues its query, it is addressed to "an interested entity", because the message consumer does not actually know the receiver (1). The query is intercepted by a discovery service (2), and forwarded to an overlay network manager in response to its query (3). After receiving the query, the manager decomposes it into several simpler ones. The sub-queries are related to:

- contracts required from raw message sources,
- message processing nodes,
- message channels to be created between the message processing nodes.

The sub-queries related to raw message sources are used by the overlay network manager to consult a query-offer matching module in order to get references of the message sources (4). The matching module will in turn query the discovery service for registered contract offers (4.1). The task of the matching module is quite complex, because it may include interpretation of semantic descriptions of message streams. In such case the module will act as a resolver of concept-based addresses. Note that separating query matching from other phases of overlay network creation makes it possible to use multiple matching modules (e.g. based on various query languages) in a single framework instance.

In order to deploy message processing nodes, given that there is no requirement for the node containers configuration to be identical, the set of appropriate deployment targets must be determined (5). The search is guided by deployment-related requirements specified by the message consumer. After checking that the deployment is feasible (i.e. there exists at least one container for each processing node to be deployed), the overlay network manager may initiate the generation of processing nodes that includes generation of needed source code and code compilation (6). Note that the message consumer does not know the interfaces of framework instance internals and therefore cannot submit a complete component. The generated code acts as a bridge between user code and containers. After compilation, the code is stored for reuse (6.1).

Once all message processing nodes are generated and each of them has at least one possible deployment target, the overlay network manager deploys the nodes (7) on containers selected according to its deployment strategy. As a result of deployment

operation, the deployment subsystem returns a set of references of running message processing nodes. These references along with message sources' ones and specifications of links are submitted to the binding service (8) to create appropriate communication channels.

Listing 1 contains the overlay network creation algorithm expressed in a more compact way.

```
loop_forever:
  ConsumerQuerySequence cqseq = DiscoveryService.getConsumerQueries();  (2)
  if( cqseq==null || cqseq.size()==0 ) goto loop_forever;

  foreach( ConsumerQuery cq : cqseq ) {                                 (3)
    MessageSourceSelectOperation[] sso = getSourceSelectOperations(sq);
    MessageSourceAdvertisement[] msadvs =
            MatchingService.findSources(sso);                           (4)

    ProcessingComponentSpecification[] pcspecs =
            getProcessingComponentSpecifications(cq);
    ContainerConfigurationAdvertisement[][] contadvs =
            DeploymentService.findContainersFor(pcspecs);               (5)
    ContainerConfigurationAdvertisement[] selcontainers =
            selectContainersFrom(pcspecs,contadvs);

    CodeReference[] coderefs =
            ComponentGenerationService.createComponents(                (6)
                selcontainers,pcspecs);
    DeploymentService.deployComponents( coderefs,selcontainers );       (7)

    LinkOperations lops = pcspecs.getLinkOperations();
    BindingService.linkNodes( lops,selcontainers );                     (8)

    DiscoveryService.publish( createOverlayNetworkAdvertisement(cq) );
  }

goto loop_forever;
```

Listing 1. Operations performed by the overlay network manager when creating a new overlay network.

One of the references submitted to the binding service refers to the message consumer that originated the query specifying the newly created overlay network. The service will therefore connect the message consumer to the network, thus letting it receive the required messages.

An important question is: what to do if some (or all) of the elements required to complete the query are missing? The answer is not simple, because the missing elements can be unavailable for several reasons, including e.g. temporal disconnection. On the other hand it is possible that the missing elements do not and will never exist. At least two approaches to handling queries that cannot be satisfied immediately exist:

- discard the query and inform the issuer,
- keep the query until all needed elements are present.

Because of peer-to-peer networks volatility which could lead to temporal disconnections of parts of message processing system, the second approach is chosen. In

that way, the framework allows queries to be issued e.g. before the sources of messages become available. Note however, that in order not to waste resources, the message processing overlays should not be constructed until all the needed elements are found. In other words, the overlays should be constructed in three phases:

- discovery of needed elements (message sources, containers for processing nodes, etc.),
- generation and deployment of processing nodes,
- creation of message channels.

By waiting with generation and deployment of processing nodes until all the needed elements are found, the framework instances would save processing power. By waiting with creation of message channels, they save network bandwidth. Moreover, in order not to assure message distribution graph coherency, the communication channels should be constructed starting from the message consumer and ending at message sources.

Get available message streams (Consumer).

Before issuing a query for messages, consumers might be interested in knowing what contracts are offered by registered producers. In order to retrieve the list of offered contracts, a consumer needs to introspect the system through appropriate interface. Information resulting from the introspection might be useful e.g. when the queries are constructed interactively. When asked for the descriptions of available message streams, the system should respond with a list of producer-originated advertisements (Fig. 20). Note that because the knowledge of available message streams and contracts may be distributed, the message consumer should expect to receive many responses.

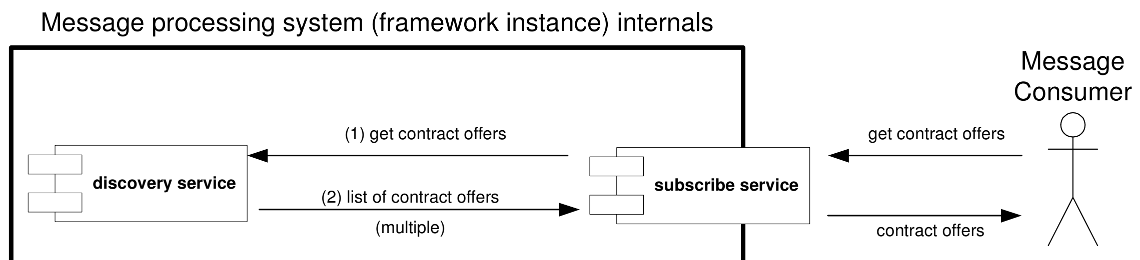


Fig. 20. Consumer-initiated introspection regarding contract offers (simplified).

3.2.2. Operation of key internal services

This section presents key functions of modules specified in the use case analysis. Because the main use case is related to creating an overlay network in response to a message consumer's query, the order of modules presentation reflects the sequence of operations taken by a framework instance when constructing the overlay. Note that in order for the instance to be fully functional, all the modules discussed in the section should be active before a query processing takes place.

Overlay network management service

The overlay network management service's task is to coordinate all the steps required to create and maintain overlay networks (see Listing 1). It is responsible for:

- interpreting the queries originated by message consumers,
- defining tasks for other modules based on the queries contents,
- executing the defined tasks in a proper order.

Overlay network management service consists of overlay network managers. Because the overlay network manager is of highest importance for the framework instance

operation, it is likely that multiple such entities exist in the instance. In such case, in order not to duplicate overlays, a scheme for sharing responsibilities between multiple managers must be designed. Such scheme may be quite simple – because most of the time-consuming operations (e.g. message processing node compilation) are delegated to the outside modules, a conceptually simple hot-standby scheme should be sufficient. The redundancy mechanisms are discussed in more detail in section 3.4.

The task of overlay network manager does not end when an overlay is deployed. After deployment, the manager is required to monitor the network in order to detect failures, tune up its parameters, etc. It is also the entity that decides to destroy the network when it is no longer needed.

Matching service

After receiving a query containing overlay network specification, the overlay network manager needs to break it down into smaller queries regarding respective steps of processing. Because finding of raw message sources is crucial for message processing overlay network creation, sub-queries regarding the sources should be processed first. Because many addressing schemes may be used in particular framework instance (even in parallel), a need for "common denominator" for addressing emerges. Because the details of processing are not expected to be interpreted by human users, reference-based addresses seem to be an optimal choice.

The matching module plays a role of address mapper between conceptual (or other) addressing and reference-based addressing. Because interpreting conceptually addressed queries requires the mapper to have an instant access to a potentially large knowledge base, it is unlikely that matching would be performed in a totally distributed manner. Rather, it should rely on a few synchronized address resolvers, the query would be forwarded to. The operation of matching service, including resolving of a conceptual address into a reference-based one is depicted in Fig. 21.

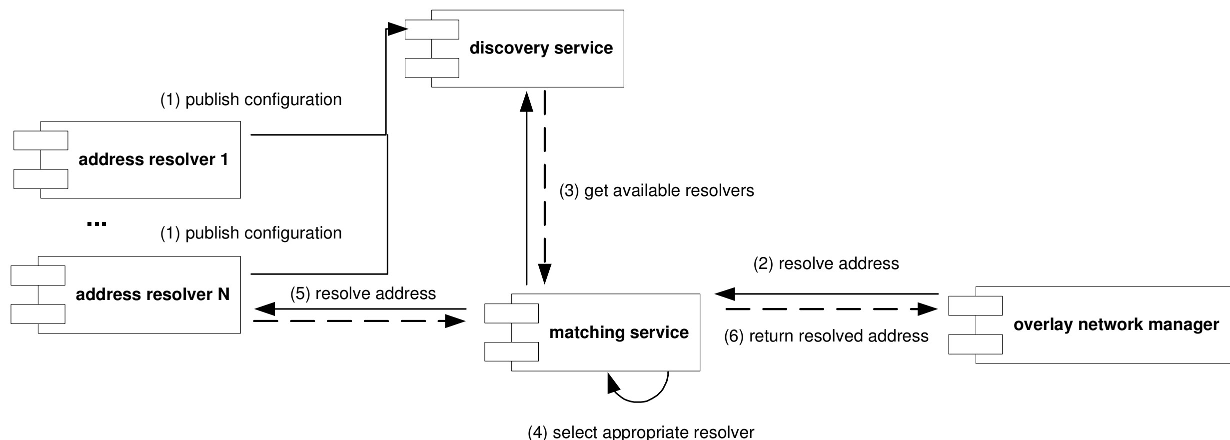


Fig. 21. Matching service operation (simplified).

The criteria for selecting a resolver to be used by the matching module would for sure include compatibility with addressing scheme and query language (if applicable). The list may be extended e.g. by parameters obtained from the monitoring module.

The underlying database is crucial for functionality of an address resolver. In most cases, the database will be built from advertisements published by message sources on behalf of respective message producers. The advertisements would be accessible through the discovery service. Some resolvers may choose to query the service when a resolution

request is received, while others would prefer to cache the advertisements locally in order not to make the requestor wait for data collection. Each of the approaches has its advantages and disadvantages. If the top-level address carries data that could be used for restricting the set of advertisements returned by the discovery service, the first approach seems reasonable. For example, topic-based address resolvers could operate that way. On the other hand, if the query submitted to the discovery service cannot be restricted by top-level addressing data, and a lot of advertisements is returned by the query, pre-caching and pre-indexing seems to be more promising.

Component generation service

In order to be deployed in a container, the components must implement certain interfaces. Although the task could be left to the end users, it would be more convenient for them if the common functionality (regarding e.g. construction of message channels) was generated by a framework service. More importantly, generation-based approach would allow for changes both in the interfaces and their implementations as the frameworks evolves. Moreover, generated code is a good place for introducing monitoring and security mechanisms based on method call interceptors. Therefore, component code generation module seems to be useful.

Generation and compilation of source code may be conducted either before the query resulting in creation of overlay network is issued or at the time of query processing. As denoted in Listing 1, the resulting compiled component code depends not only on the user-provided specification, but on the result of target container selection as well. Because REMP supports containers of various configurations, components deployable in one kind of containers may or may not be deployable in another. Therefore, although preparing a component prior to issuing the query may result in faster deployment and startup of the network, taking such approach requires the user application to know framework internals and use them directly. Therefore, the latter approach, in which some internal entity (i.e. overlay network manager) takes care of the generation seems to be preferable (Fig. 22).

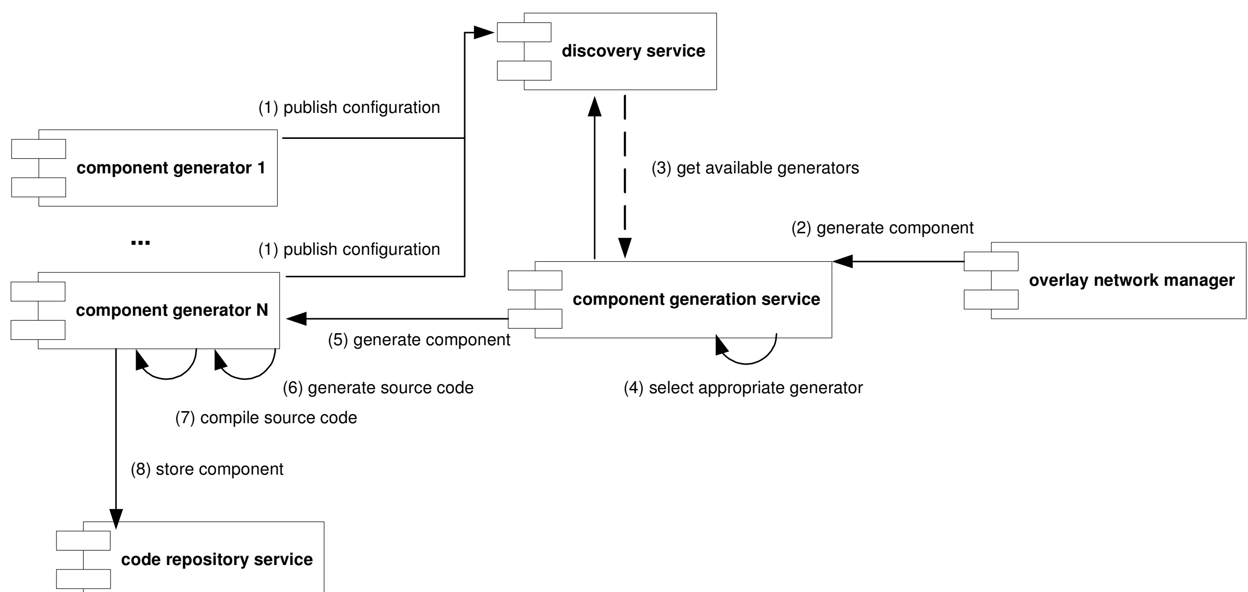


Fig. 22. Generation of a message processing node's code.

In order to support a broad range of instances, the framework should not be bound to a single source code language. Rather, the entity responsible for code generation and compilation should look for a module capable of generating code for processing node

written in a particular language and suitable for selected type of containers. Dynamic discovery module can be reused for that purpose.

Language and container compatibility are the basic criteria of selecting node generators, but the component generation service could also take into account other factors, such as compilation queue length, administrator-defined preference level, etc.

Code repository service

In order to support reuse of generated code, a repository should be provided. The repository may be useful in at least three cases:

- some overlays share processing steps, but do not run simultaneously,
- due to optimization or environmental changes (e.g. a disconnection of a node container) a part of an overlay needs re-deployment,
- a user application compiles and stores the code prior to issuing a query.

The proposed repository is based on multiple distributed storages. For failure resiliency reasons, codes are stored in at least two places. Fig. 23 illustrates the process of storing component code.

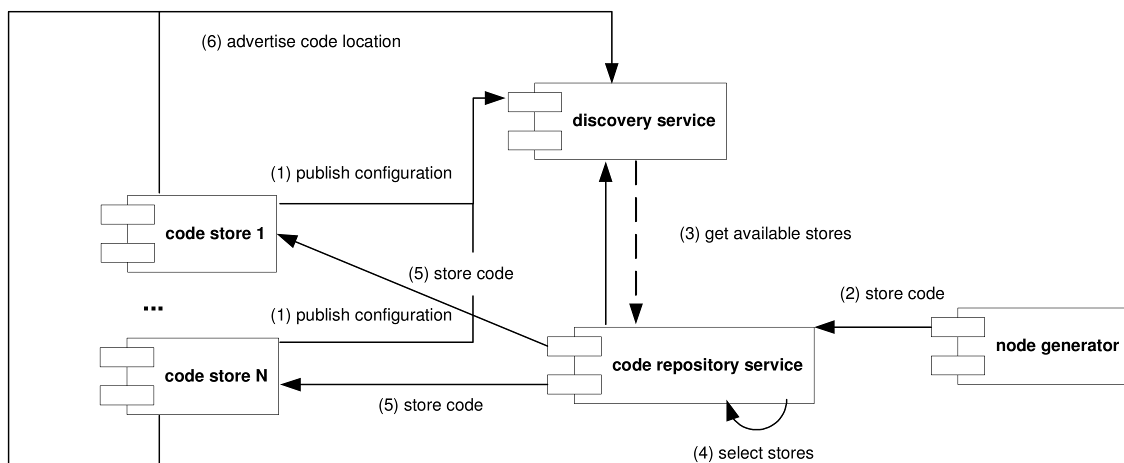


Fig. 23. The process of storing generated component code (simplified).

As the last step of code storage procedure, the stores publish advertisements regarding locations of the stored codes. The goal is to help the containers that need to download the components' code communicate directly with the stores.

Deployment service

The deployment module gathers information about containers available in the framework instance and presents options for deployment of components. The deployment strategy is key for overlay networks' efficiency. Moreover, the strategy may depend on more than efficiency-related factors, including e.g. privilege levels of query issuers etc. Therefore, in order to keep the module simple, the deployment policy should be implemented by external modules.

The deployment module offers two functions to its users (Fig. 24). It is capable of searching for containers compatible with particular component (possible deployment targets) and executing the deployment and activation of components. In order to conduct the search, the deployment service needs to know the configuration of available containers and requirements of the component to be deployed, that should be included in the component specification provided by message consumer. The deployment process is split

into two phases (selection of possible containers and deployment of code), mainly because the results of the first phase are used by component generation service. As pointed out in Listing 1, the list of configurations of containers compatible with the component to be deployed is passed to the component generation service.

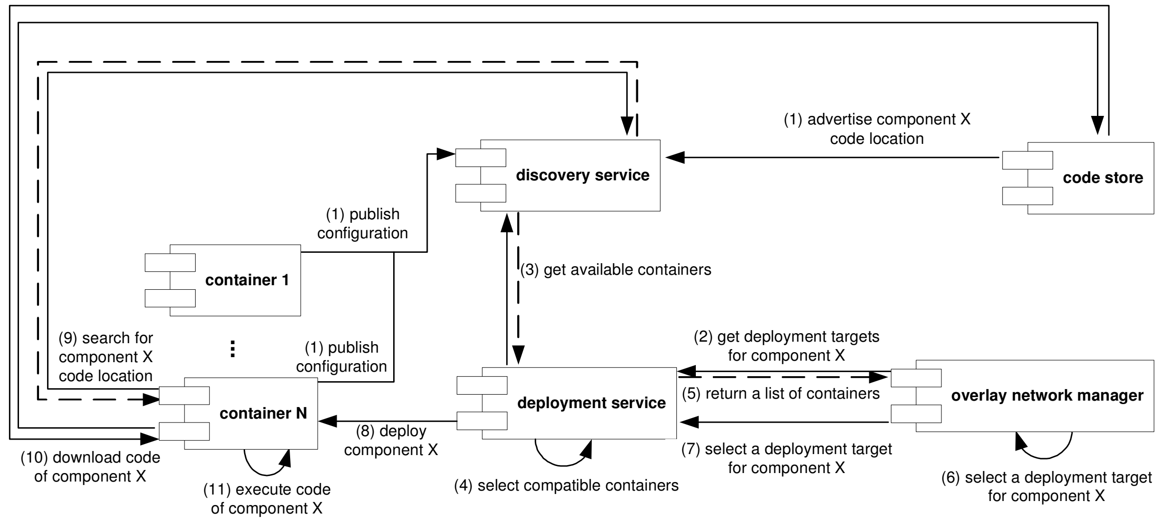


Fig. 24. Deployment and instantiation of a message processing node.

Containers are the most important elements of framework instance infrastructure. The deployed components' capabilities depend on the services offered by the containers heavily. However, typical message processing nodes do not present severe requirements to the containers – they need only to have access to the inbound and outbound communication channels. However, the proposed solution does not allow direct access to the channel. Instead, the channels are accessible through a container-provided interface. In that way, the framework provides for easy introduction of intermediaries, such as message interceptors or security checkers.

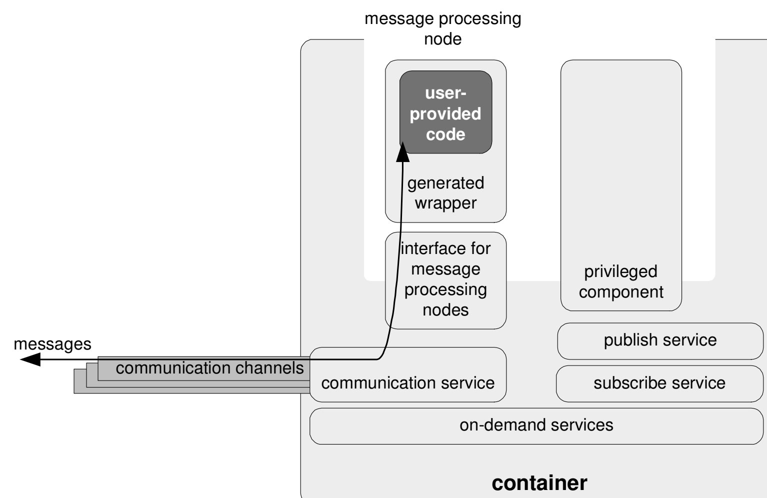


Fig. 25. A basic framework instance container.

If the framework instance is heterogeneous, i.e. it contains code generators for various execution platforms, compatibility between the executable code to be deployed and the deployment target (i.e. the container) needs to be checked not only by the deployment service proxy, but by the worker (i.e. container) as well. The compatibility check ensures that the deployment of a component will be successful, i.e. it will fulfill two conditions [Belguidoum 2007]:

- the system will provide the resources and services required by the component,
- the component will not conflict with already installed instances.

The examination of the component to be deployed may be extended beyond simple properties checking. For example, in case of using spontaneous containers, the examination may result in downloading additional modules needed by the component.

The role of the containers does not end with handling message processing nodes. Some of the framework internal entities may also be implemented as components in order to be deployed when needed. As described in earlier sections, such components would require access not only to the communication service, but to other as well. As an example of such component consider an instance of overlay network manager, which would require access to most of the services that build up the framework. In order not to make the containers too heavy, the services should be downloaded and instantiated on (privileged) component's demand.

Binding service

The task of binding service is to create a set of communication channels that interlink the previously deployed components. In order not to make the components buffer the messages, the channels should be created in direction opposite to message flow, i.e. from the message consumer to producers. The binding service needs to communicate the message sink (which represent message consumer), message sources (which represent message producers) and containers, which act as both message sinks and sources. Therefore, all the mentioned entities must implement appropriate protocols.

Fig. 26 illustrates creation of a single message channel.

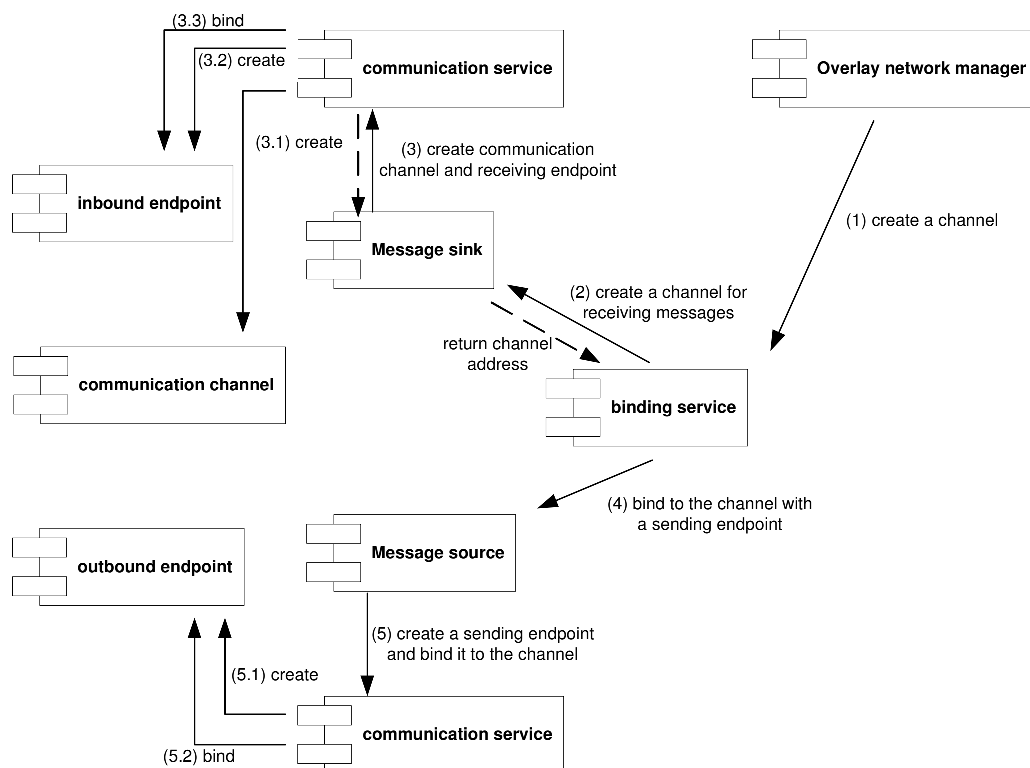


Fig. 26. Creation of a communication channel.

Note that in order to bind to the same message channel, both the message source and the message sink must share metadata regarding addressing. There are two possibilities

for organizing the addressing. First, similar to the TCP way of connection establishment, relies on letting the message source know the reference of message sink and a reference of message channel ("socket"). Second is to assign an address to a channel itself and share the reference of the channel both to the message source and sink. The second approach seems to be more flexible, because in case of any component migration, the re-binding operation applies only to the migrated side. Note however, that although conceptually simpler, the second approach makes the underlying middleware more complicated.

Dynamic discovery service

Milojicic et al. define the class of peer-to-peer systems as "a class of systems and applications that employ distributed resources to perform a critical function in a decentralized manner" [Milojicic 2002]. According to Coulouris et al., peer-to-peer systems are "distributed systems and applications in which data and computational resources are contributed by many hosts on the Internet, all of which participate in the provision of a uniform service" [Coulouris 2005].

The perspective of aggregating a vast amount of resources available at "the edge of the network" for accomplishing e.g. a computational target was the driving force for creating environments that enable peer-to-peer networking. Most of contemporary peer-to-peer networks make use of dynamic discovery of resources. The discovery services are implemented in various ways, starting from centralized repositories (either replicated or not), through various distributed hash tables, to completely distributed schemes, in which neither resources nor indexes are shared.

Because totally distributed discovery services tend to be communication intensive, using such service extensively could lead to unacceptable bandwidth consumption and in turn limit framework instances' scalability. Centralized (and replicated) or hybrid approaches seem to be a better choice. Note that from the proposed framework point of view, extensibility is more important than the organization of the service itself. Because most of the modules related to the key use case use the discovery service extensively, the ability to define custom types of advertisements and custom indexes is of high importance.

Communication services

Framework instances should benefit from two types of communication modes:

- connection-oriented (message channels),
- connectionless (simple query/response).

The connection-oriented service is expected to implement the message channel abstraction above a real network. Although the publish/subscribe system can be constructed upon any kind of such service, it would be easier to implement if the service provided by the framework supported reliable channels. As discussed earlier in this section, it would be desirable if the service assigned addresses to channels, not only to their endpoints. Moreover, support for many-to-many communication would make the construction of overlays easier and provide for more readable overlay specification.

The query/response service may be useful for implementing simple data exchanges that occur between framework instance internal entities. As an example consider address resolution. From the overlay network manager point of view, the only data it sends to the matching module is the address to be resolved and the only return value is the resolution result. The overhead related to connection establishment seems unreasonable in such case.

3.3. *Overlay network maintenance*

Peer-to-peer overlay environments use a communication model that is much different from traditional client-server environments. The communication between peers is more spontaneous and relies on creating short-term unidirectional or bi-directional connections on demand. Peer-to-peer systems are decoupled and should be aware of the possibility of any peer getting disconnected at any time. In context of message distribution systems it means that the system could not rely on any peer's presence (including the peers its internal entities are running on). Therefore, the task of a message processing system does not end when a message processing overlay network is deployed. After deployment, the network must be maintained in order to:

- ensure proper operation of network entities,
- evaluate and introduce modifications of overlays to improve their operation.

In general, no knowledge of semantics of operations performed by overlay network nodes can be assumed. Therefore, governance of proper operation is restricted to keeping track e.g. of availability of the nodes, values of quality of service parameters, etc. The evaluation of options for overlay network modifications must also be based on the same set of criteria. Nonetheless, it seems reasonable to provide (extensible) subsystems to support even the basic functionality. Therefore:

- a monitoring and metering service should be included in the framework,
- the overlay network manager should be able to perform modifications upon a working overlay.

Monitoring and metering service

The goal of the monitoring and metering service is to provide its user with:

- values of current and past variables illustrating the state of an overlay network or framework instance entities,
- means for executing runtime measurements.

As an example user of the runtime measurements, consider an overlay network manager that received a notification about newly attached container. In order to decide, which message processing nodes should be migrated to the new container, the manager needs to know, whether the migration will improve performance of particular overlay network. Therefore, it instructs the metering service to perform a series of round trip time measurements between the nodes on the current and possible paths. Once the measurements are completed, the manager gets notified about their results and is capable of taking the decision. A single measurement sequence is illustrated in Fig. 27.

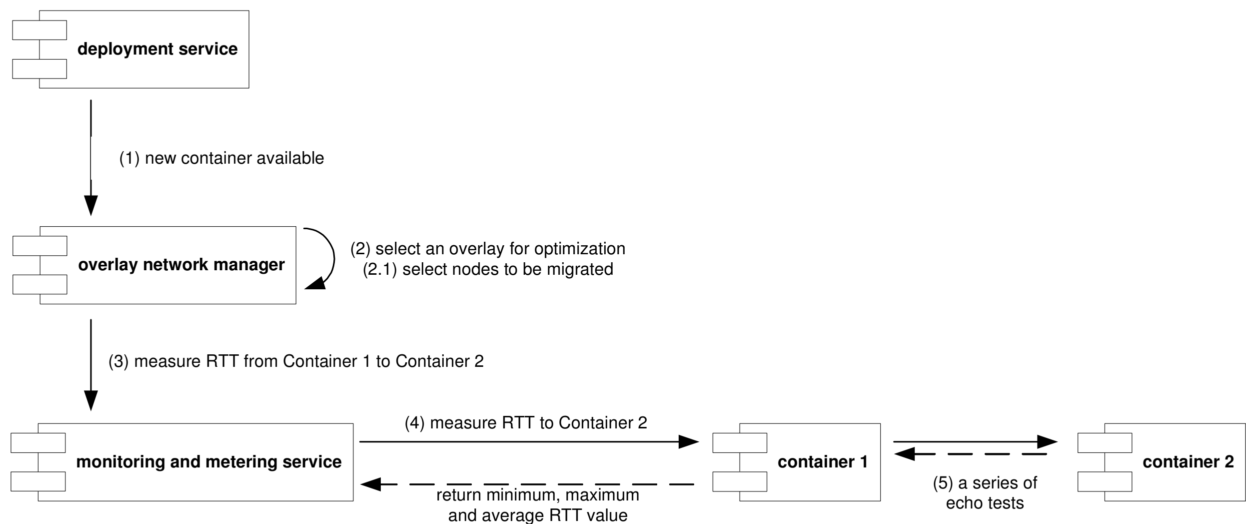


Fig. 27. A simple use case for the monitoring and metering service.

Failure recovery

The modifications of overlay networks may result from various reasons, including disconnection of part of the overlay. The case has a few sub-cases, including disconnection of:

- intermediary processing nodes,
- message source(s),
- message sink,
- overlay network manager.

In case of part of the network getting disconnected, the overlay network manager needs to re-deploy the missing intermediate entities. If the entities are stateful, recovery means (based e.g. on checkpointing service) should be used. After re-deployment, and re-linking the overlay network should be operational. Note that thanks to the code repository service keeping the compiled code of processing nodes there is no need to go back to the component code generation phase. Of course, in case of severe failure, the manager might be forced to re-create the whole network.

In case of a disconnected message source, there are three possibilities: to find an equivalent message source (by using matching service), to disregard the event or to close down the overlay network. The decision should be taken by an overlay network manager based on message consumer's query. If a message source is not very important, the message consumer should mark it appropriately. In other case, after trying to find an alternative, the manager should close down the network.

It would seem that disconnection of message sink should result in closing down the overlay network immediately. After all, supplying the message sink with processed messages is the reason for keeping the overlay network running. However, because the disconnection could be temporal, and the re-creation of an overlay time consuming, the overlay network manager should refrain from closing down the network for some time. The default value of the timeout should be specified by framework instance configuration or by message consumers. Note that in many applications, the message consumer might be connected only from time to time, e.g. for gathering daily reports.

The case of disconnected overlay network manager seems to be the hardest to detect and repair mainly because the manager does not belong to the overlay network and its disappearance does not break down the processing. However, because a single manager

can manage multiple networks, its presence is crucial for the framework instance as a whole. The services that help to recover from network manager failure are described in the following section.

Adaptation

Framework instances may choose to implement various adaptation strategies for overlay networks. In order for the developers not to build adaptive instances from scratch, migration of a message processing node from one container to another as one of the most important mechanisms that should be provided by the framework. Assuming that the processing nodes are stateful, the migration process can be implemented as a sequence of steps illustrated in Fig. 28.

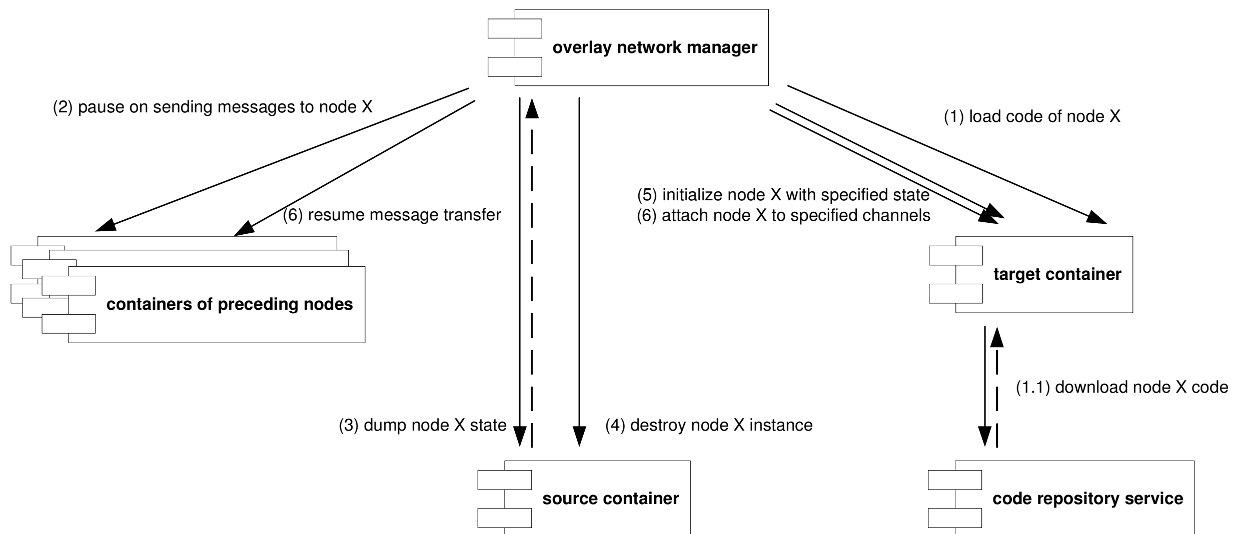


Fig. 28. A simplified node migration process.

In the depicted scenario, in order to shorten the time of migrated node unavailability, code loading is separated from migrated component instantiation. After the code is loaded, the components (either message sources or message processing nodes) that were sending messages to the migrated node are requested to pause the stream. That would require them to buffer the messages to be sent until the migrated node is active. Then the state of migrated component is dumped into a component-specific record that is transferred to the target container. Just after the state is received by the overlay network manager, the no longer needed copy of the migrated component is destroyed. Next, the new copy is initialized with the state record and attached to the communication channels. After that, the predecessors are signaled to resume message transfer.

Similar sequence of operations may take place when the message processing system chooses to change the implementation of particular processing node. Such decision may be taken e.g. if a new (presumably better) implementation of a particular component becomes available. If only the interfaces (input and output streams) of the processing node are preserved, the change could be conducted without destroying the whole affected overlay network.

Allowing the applications for tuning parameters of message processing nodes is a real issue, because of security and reliability reasons. It seems that most of the “interesting” parameters are bound to message channels. As an example, consider the minimum inter-message interval. Because the channels would be connected to the containers, not components, having access to the container a malevolent overlay could disrupt services

provided by others. Therefore, it seems reasonable to forbid access to the system internals for ordinary applications and leave the possibility of parameter tuning only for overlay network managers.

3.4. Framework instance maintenance

Because wide area peer-to-peer networks are rarely fully predictable, the framework needs to be equipped with maintenance means that would react to environmental changes. The list of basic task related to maintaining the framework instances includes:

- keeping track of available resources,
- reading current state of entities building up the instance,
- logging states of key entities both for diagnostic and recovery purposes,
- replicating the core framework instance nodes and keeping their states synchronized,
- ensuring security of internal operations.

Note that most of the tasks should be performed continuously, but in different contexts. For example, only the entities that are capable of deciding where to deploy a particular component need to be informed about available containers.

Introspection

In order to perform introspection in distributed systems either a storage for system state log or dynamic discovery service should be supported. State log require the entities present in the system to keep their records updated. However, because in changing runtime environment, the log may become disconnected from a part of the system, its consistency becomes an issue, so replication and synchronization mechanisms regarding the logs should be considered. Moreover, a sudden disconnection of a logged entity results in its inability to update its record. Therefore, keepalive mechanisms for each logged entity should be also implemented. Considering that introspection is used infrequently, the overhead related to keeping the logs consistent seems to be unacceptable. Note that logs could be useful, but for different purposes, such as crash recovery.

Dynamic discovery service operating on a distributed database of state records that describe the current state of a system seems to be a better solution for supporting introspection. Discovery services based on distributed hash tables share indexes of the database while keeping the indexed resources (state records) at the described entities. Such approach provides for better consistency between real and discovered system state, because updating is much simpler (no synchronization is needed) and if the described entity gets disconnected, so does its state record.

Dynamic discovery service is expected to be the main means of introspection for the framework instances. It should provide its users with information about:

- the existence and configuration of internal system entities,
- the contracts offered by registered message producers,
- the unresolved queries issued by message consumers,
- the components stored in the code repository and code locations,
- the existence, specification and implementation of overlay networks.

More specific information regarding the overlay networks should be available at containers that host particular components.

Reflection

In order to stay operative, any framework instance would need to assure a minimum number of entities of various kinds keep running. As an example consider a minimum of three component code storages, two log servers, three different code generators etc. By implementing them as components, the framework would offer its instances the possibility to deploy the needed entities dynamically, similarly as in the case of overlay network components. However, because the internal components have higher requirements (in terms of needed services) than message processing nodes, they should be easily distinguishable from the nodes.

Fig. 25 illustrates the idea of privileged component. By using techniques such as code signing, the message processing system developer might easily mark the more privileged components. A container that receives such component would in its turn check the signatures in order to classify the components and grant appropriate privileges regarding e.g. access to selected local instances of REMP services.

Dynamic instantiation of required entities, although important, is not the only reflection mechanism that could be used by the developer and administrator. Similarly to the adaptation case:

- the implementations of particular modules could be changed at runtime,
- the runtime parameters of internal entities can be dynamically tuned.

Parameter tuning is even more important than in case of adaptation mechanisms, because reflection is expected to perform it continuously, although in most cases indirectly. When using several equivalent entities, such as address resolvers, metrics describing their state and preference levels should be introduced. Such metrics might include task queue length, average time to complete a task, time from last disconnection, etc. Based on the measurements, the servant entities could either be assigned a metric value or calculate it itself. For example, based on the average time of processing and task queue length the code generator can calculate its preference level that would be used by proxies to balance the load between generators.

Checkpointing

The recovery mechanisms are based on logging the states of the entities. The repositories of states should be kept in a place that is accessible to all the entities that are interested in preserving the data that could be used for recovery. In a large-scale peer-to-peer distributed environment keeping the states in a centralized place, although convenient from administrative point of view, is rather impractical, because of the probability of server disconnection. Therefore, although maintenance of distributed servers can be difficult, a distributed checkpointing service, based on the proxy-worker pattern is proposed.

At a first glance, the checkpointing service behaves similarly to the code repository service. Once submitted to the service, the states get replicated in a few redundant places. However, because checkpointing operation is expected to be repeated, the storage operation is also repeated many times. Therefore, the selection of checkpointing service worker (i.e. state log server) should not be performed at each service call but only once. Unless the entities that use the service are migrated, the selection results could be cached by their local instances (proxies) of checkpointing service. In case of component migration three ways of assuring logs consistency are possible:

- sending the results of selection (including e.g. addresses of selected servers) as context information along with the component state,
- designing a hash function that based on the component address and set of server addresses would compute the addresses to be used,
- assigning identifiers to the states and including them in the configuration advertisements of servers.

Although it complicates migration, the first option seems to be the safest. Even if the target container did not discover appropriate checkpointing service workers, given the addresses, it should be able to contact them.

The second option seems to be the most flexible one, because it allows sharing states between multiple components. However, in order for the hash function to yield the same results at different places, the set of (discovered) addresses of log servers must be identical. In case of dynamically changing environment, this requirement may not always be met.

The greatest advantage of the third option is its good support for crash recovery. If the identifiers of "logged" components are in some way advertised (not necessarily along with the state log server configuration), the work of recovery mechanisms gets simpler. However, after migration, it could take some time for the target container to discover appropriate servers.

As a compromise, a combined approach, including elements of the first and third options, was chosen. When opening a log, a component is assigned a new anycast address bound to a one-to-many communication channel. At the receiving ends of the channel, state log servers are attached. In order to support recovery, the servers include the channel address as well as the logged component address in their configuration advertisements. To speed up migrated component instantiation, the address of the communication channel is transferred along with the component state to the target container. The servers should monitor each other presence in order to keep the requested number of log copies. The operation of checkpointing service is depicted in Fig. 29.

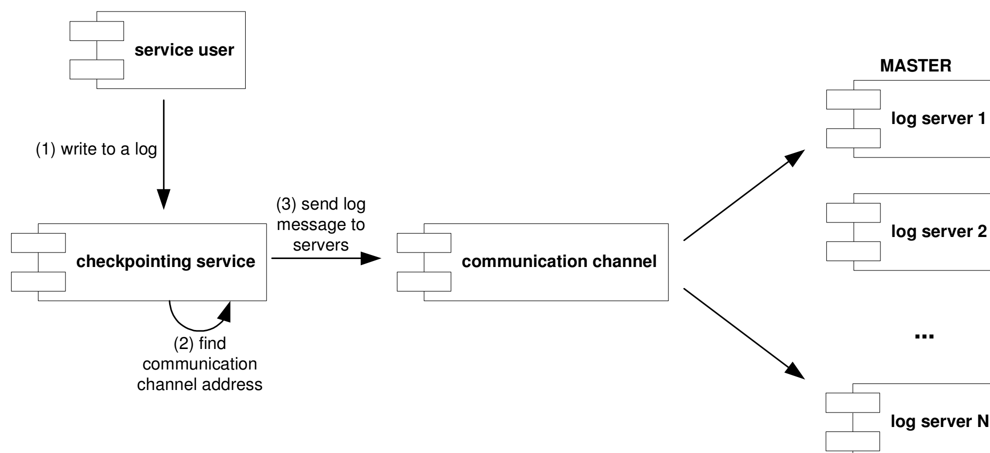


Fig. 29. Operation of checkpointing service (simplified).

In order to set up the communication channel, a master (managing) server may be elected. Because when processing first "write" operation, the service is unable to determine which channel to use (the channel may not exist), it should contact the master server through a well-known communication channel. The master server, using discovery and monitoring services would determine the servers to be attached to the state logging communication channel, instruct the selected servers to create receiving ends of the channel

and return the channel address. In order to assure there is exactly one master server, all the servers could use the redundancy service described in the following subsection.

Logs tend to be voluminous. In order to speed up the recovery of components, their "full" states should be saved periodically. Moreover, the checkpointing service should be notified, which record contains the "full" state. In such case, it would be able to mark the log record as the starting point for recovery. If needed, the component instantiated by a recovery mechanism would be assigned the last "full" state and receive a stream of only those incremental state updates that were saved after the update.

Redundancy and load balancing

Service redundancy is crucial for dynamically changing environments. From the framework's point of view, redundancy of instances of internal services responsible for construction of message distribution graphs is of particular importance. The proposed framework uses two approaches for management of redundant instances:

- proxy-worker pattern,
- redundancy service.

The proxy-worker pattern (Fig. 30) is used by services that submit batch jobs to specialized entities, e.g. matching service, code repository service. The proxy part of the service is located on requesting entity and is responsible for:

- collecting configuration advertisements issued by workers,
- gathering monitoring information regarding the workers,
- selecting an appropriate worker for particular jobs,
- issuing requests and collecting responses from the workers,
- informing the requestor of job status.

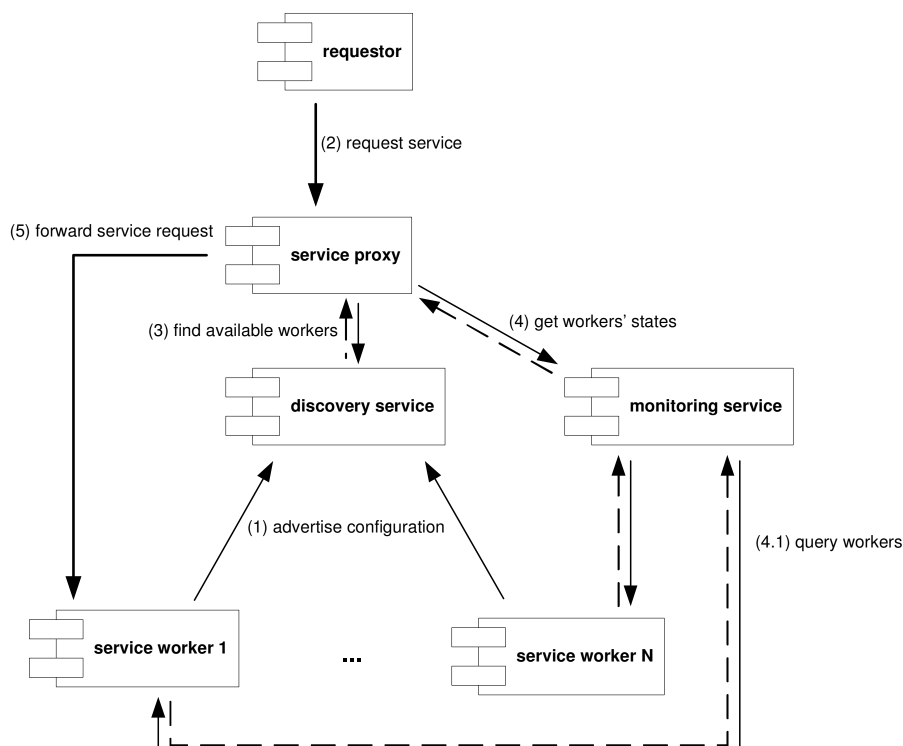


Fig. 30. Worker selection in services based on the proxy-worker pattern.

The drawback of the approach is that the proxies do not collaborate with each other in order to make better use of workers. Instead, their decisions are based on the discovery

and monitoring services reports regarding available workers. That could lead to worker usage synchronizations. Nonetheless it seems to be a rare case, because the services based on proxy-worker pattern are used only in distribution graph construction phase, and not in message processing. Introducing a new inter-proxy protocol would lead to better usage of workers, but would also introduce new states of proxies and increase the complexity of proxy operation.

Redundant overlay network managers are an example of a case, when the redundancy of service instances must be managed by themselves. The managers form a group of common interest. When a member of the group fails, some other entity takes over its responsibilities. Group members can also perform load-balancing in order not to simply back up one another, but to provide more efficient service. Although the framework currently does not include another such group, tight coupling of the redundancy management functionality and managers seems to be a poor decision in the context of the emphasized need for framework extensibility. Therefore, a separate redundancy service is proposed.

There are many approaches in the IP world that implement such functionality between routers. Those include: Hot Standby Routing Protocol [Li 1998], Gateway Load Balancing Protocol [Cisco 2005], Common Address Redundancy Protocol[OpenBSD 2003] and Virtual Router Redundancy Protocol [Hinden 2004].

HSRP and VRRP allow multiple routers to create a backup group that uses a virtual IP address. The virtual IP address is configured as a gateway for a group of hosts. By responding to the ARP queries one of the routers, elected to be active, directs the traffic originating at the hosts to itself. The active member of a HSRP or VRRP group informs all the other members about its presence by issuing periodic messages to a well-known multicast address. If – for a specified amount of time – there is no message from the active router, another one takes over the virtual address. In addition, in order to speed up convergence, HSRP routers elect a standby router, that is the first to take over the failed active router's responsibilities. CARP is an open VRRP-like protocol successfully used e.g. for firewall redundancy[Raspet 2004][Artymiak 2004].

Similar functionality can be implemented in overlay networks. The backup groups can be created e.g. by redundant instances of a service needed by many clients. The entities can use a well-known multicast communication channel to communicate with each other in order to elect the active one, announce its presence and detect its disappearance. In order to provide transparent redundancy, the entities forming the redundancy group can be configured with a virtual address advertised by the currently active entity. However, assuming that a communication channel is an addressable entity, it occurs that no virtual address is needed. The requesting parties might be given only a channel address without knowing the receiver of requests, which happens to be the currently active entity (Fig. 31). From the requestor's point of view, the channel address can be interpreted as an anycast address to one of the equivalent service instances.

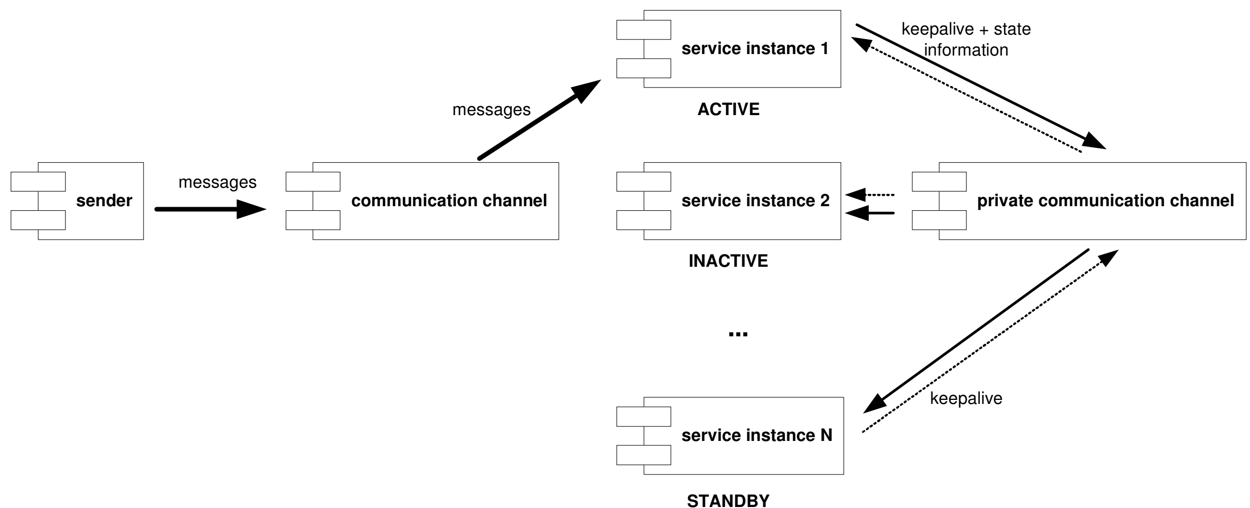


Fig. 31. Cooperation between redundancy service enabled instances and user message flow.

The GLBP-like approach is harder to be implemented in an overlay network primarily due to its complexity. GLBP elects an active virtual gateway (AVG) similarly to HSRP. The AVG is then responsible for managing other group members, called active virtual forwarders (AVF) by assigning them virtual addresses to be served. In overlay network, in order to choose the AVG, the service should take into account more factors than GLBP operating in a LAN environment does. These include: latency of links between end peer and redundant services, end peer's bandwidth requirements, end user quality of service requirements, etc.

Therefore, a compromise between the pure HSRP-like and GLBP-like approaches, similar to multigroup HSRP was chosen. The redundant entities – either framework internal services or message processing nodes – may reside on containers. Just like in case of multigroup HSRP, different containers may run active copies of different modules, balancing the overall load between them. Fig. 32 illustrates the approach in the simple case of two redundant services running on two different hosting peers.

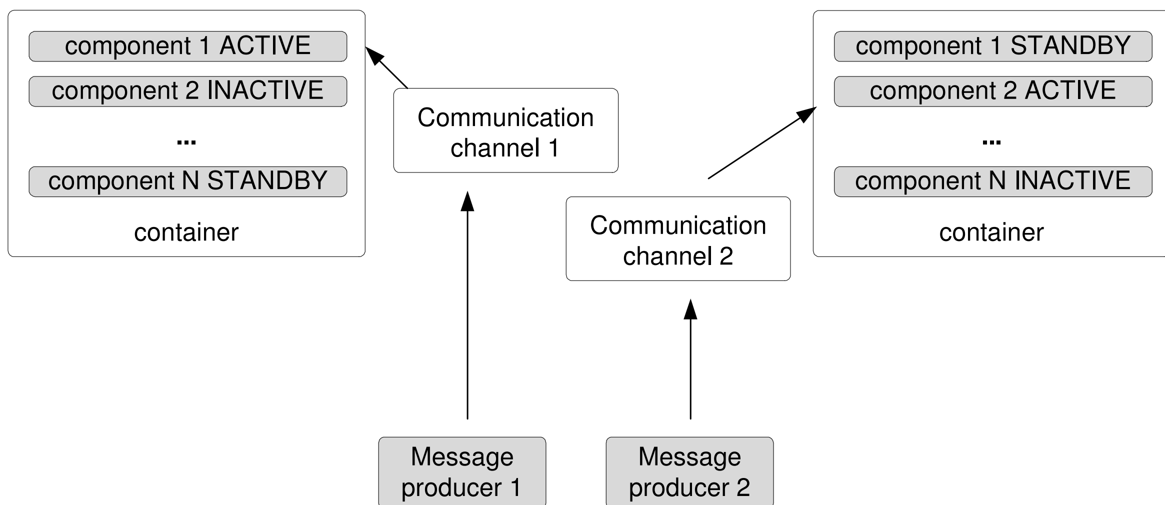


Fig. 32. Transparent load balancing using redundancy service.

More detailed information regarding the chosen mechanism for electing the active instances is and synchronizing their state with the standby ones is described in section titled "Redundancy Service, Service Redundancy Protocol" of Chapter 4, which describes prototype implementation.

3.5. Framework instance extensibility

The publish/subscribe pattern is used in a variety of applications. Because it is not possible to envision all applications of the proposed framework, its architecture and implementation must allow for changes and improvements. In order to satisfy this requirement, the framework's architecture should be modular, with well-defined interfaces not only for framework instance users, but also for developers, who want to change the default behavior of the framework internals. Allowing for changes may cause losses in terms of effectiveness. As an example consider a discovery service, which is crucial for virtually any framework instance. Discovery services in modern publish/subscribe systems frequently benefit from knowing the languages the queries and resource descriptions are expressed in. The languages are developed with discovery efficiency in mind, and are tightly integrated with the discovery services. In such systems, there is no possibility for the developers to use their own query or resource description language. Clearly, extensibility is partially sacrificed for the sake of efficiency. The proposed framework takes different approach. Although efficiency is not completely neglected, the flexibility and extensibility are of more importance.

3.5.1. Ease of integration with existing systems

In order to be able to reuse existing software, the framework instance developer would need support for standard document representations. A common way of ensuring interoperability is expressing he information passed between its source and sink as a structured document. Among many ways of encoding structured documents, the Extensible Markup Language (XML)[Bray 2008] is one that provides a great level of flexibility. There are pros and cons for using the XML for message encoding. The main drawback is the relatively high overhead of processing XML messages. On the other hand its flexibility is a great advantage. What is noteworthy, the language is recognized as an Internet standard, what makes it a good choice for software integration.

There are a number of languages used for describing structure and/or semantics of XML documents as well as associated query languages. However, note that the support for semantic description and selection of messages can be important only to the external users of the system, and does not seem to be required for internal message processing system communication. The messages exchanged between internal system entities typically have well-defined semantics and are useless from the external entities' point of view, so using semantic description language in system internals would only introduce unnecessary overhead.

The framework is expected to *support* the interpretation of message contents, not to *perform* it itself. Therefore, its architecture should be modular, with interfaces for different message interpretation engines that can be based on various message semantics description and query languages. Note that semantic description which defines what the message contains must be accompanied with message syntax description that defines where exactly the information is placed inside the message in question.

3.5.2. Framework customization

REMP instances come as results of the process of installing, customizing and configuring framework elements. While the installation and configuration parts are closely related to the framework implementation, most of the customization options can be listed

based on the framework instance model, which is the main topic of this chapter. Therefore, this section is dedicated to list the main customization options related to various parts and aspects of REMP instances.

Containers. The functionality offered by framework instances depends heavily on the functionalities offered by containers that form the most important part of their infrastructures. Therefore, customization of containers is the key to providing functionality extensions. As it was mentioned in section 3.2.2, framework instances allow various containers to be used, provided that they advertise their configurations, that are matched against the requirements of components to be deployed (Fig. 24).

The specification of particular component's runtime requirements may contain a set of services the component needs to be executed. For example, if a component was expected to serve as a dynamically created message source for overlay network processing reuse purposes, it would need access to a Publish Service instance. In case of more complex components, more than one services may be needed. In order to increase deployment flexibility, the containers should be dynamically extensible, i.e. should be able to download and instantiate the needed services in case they are not instantiated at the moment the component code is downloaded.

Languages. The framework is designed to operate in a multi-language environment. This applies to message semantics and syntax description languages as well as to message processing node source code languages. Through the use of container configuration advertisements and component compatibility statements, diversity of containers is also allowed. A message processing system developer might introduce e.g. a new source code language by implementing appropriate component generation service servant. Proxy parts of the service operate on configuration advertisements and do not need to be changed.

Query address resolution. Concept, topic and reference based addresses are allowed to be used simultaneously. A user that wants to introduce a new query addressing scheme would be required to implement an address mapper that translates the new addressing to the internal reference-based one. The mapper would be fed with advertisements containing contract offers registered by message sources and queried with respective parts of overlay network specifications issued by message consumers.

Overlay network management. Depending on the purpose a particular framework instance is going to serve, the overlay network managers are expected to have various priorities. As an example, consider the choice of deployment strategy, which may depend on factors that cannot be determined at framework design phase, e.g. on the foreseen volume of data to be transferred.

Monitoring and metering. The service offers a possibility to execute active measurements of framework instance characteristics. Although simple checks, such as round-trip time could be provided by the framework, more specific calculations need to be implemented by the instance developer.

Code and logs repositories. The instance developer might influence the behavior of the repositories mainly by changing configuration parameters such as preference levels that need to be provided by the implementation. Because the proxy and servant parts are separated, introduction of new servants is also possible, but does not add much functionality to the developed system.

Security. Because the security-related services are going to be inherited from a peer-to-peer middleware the framework is going to extend, the options for the developer

will be determined rather by the extended middleware than the framework. However, the consumer query is expected to contain security-related part. Moreover, code signing techniques are going to be used to indicate the privileged components. The developers would be able to configure the set of trusted component sources and respective sets of services available to the components that are signed by the sources.

3.6. Summary

This chapter was dedicated to specifying the framework instance model upon the results of analyzing the requirements of users of the framework and framework instances. The analysis presented in this chapter lead to identification of services the framework should be built of as well as sets of entities needed for ensuring the operation of particular services.

The REMP framework is intended to support dynamic creation of message processing oriented overlay networks. Content-based publish/subscribe systems seem to be one of the most complex examples of such overlays. Although analysis of their capabilities, presented in Chapter 2 did not lead to identification of new needed services, it resulted in pointing out several conclusions – that refer mainly to framework services customization.

After creation of a message processing overlay network which implements the virtual producer concept, due to the possible changes in runtime environment, it is important to monitor its operation to ensure an uninterrupted service to the end user. Although in some cases the failures will be noticeable to the users, a set of services to alleviate the problem was proposed.

The stress that was put on making the framework services customizable may result not only in increased applicability area of the framework, but also in opening the possibilities regarding the choice of underlying middleware. Chapter 4 presents a REMP prototype built upon an existing middleware technology.

4. Prototype implementation

This chapter presents the prototype implementation of the message processing overlay networks deployment framework, that was created to verify the concepts presented in the specification of REMP instance model, which was introduced in Chapter 3. The implementation was created with two main assumptions in mind:

- to make the user API simple, but expressive, and
- to make the framework customizable and extensible.

Section 4.1 describes key technologies used by the prototype implementation, as well as their influence on most important details of prototype implementation. Section 4.2 presents the minimal set of entities building up a running prototype instance. Sections 4.3 and 4.3.2 describe how the entities cooperate to implement functionality described in key framework instance use cases. Particular attention is put on overlay network creation, which is the most important use case. Section 4.4 overviews services important from the framework self-management point of view. The chapter is summarized by section 4.5.

4.1. Technologies used for prototype implementation

The framework prototype implementation is based on a set of technologies reused to provide parts of its functionality. The choice of implementation technologies results in introducing constraints on the implementation, e.g. on network entities addressing. This section is devoted to presenting the chosen technologies and their implications.

4.1.1. JXTA

JXTA is a Java-based, general purpose platform for implementing hybrid peer-to-peer networks, that was chosen as a base for framework prototype implementation. The platform is constructed upon a set of protocols implemented by services[JXTA 2007], which are partially reused by the framework. Fig. 33 illustrates the architecture of the REMP framework prototype built upon JXTA. The services that match part of functionality specified in Chapter 3, which described the REMP framework instance model, are shaded in gray.

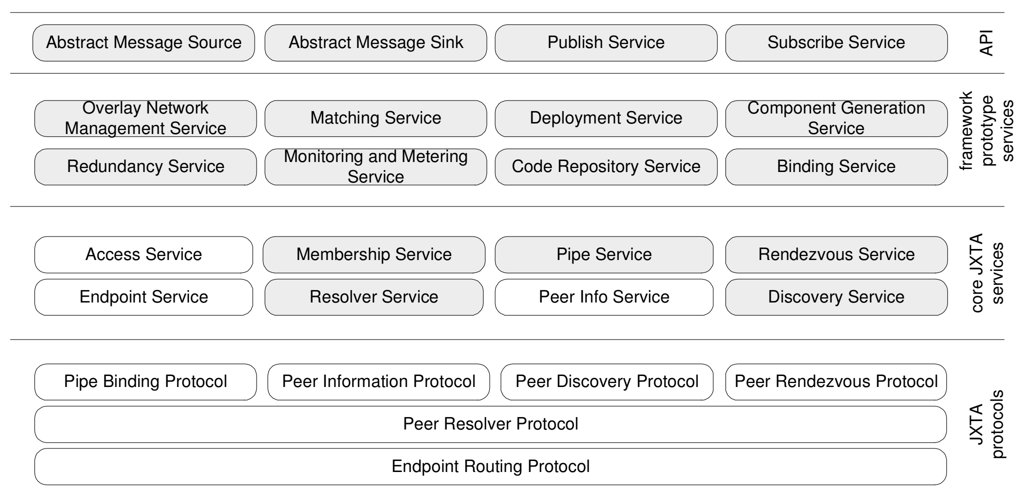


Fig. 33. Overview of framework's prototype implementation architecture.

This section briefly describes JXTA services that are used by the framework prototype implementation.

Discovery Service

The Discovery Service provided by the JXTA platform is used to look for advertisements published by message sources, sinks and internal system entities (in particular service workers). The Discovery Service is a peer group service and therefore each peer belonging to the group can instantiate and use it. The advertisements published with the service are available for peers that belong to the same group. In order to improve scalability, the advertisements are kept locally in the publisher's cache (typically disk cache) and only the indexes are propagated to the rendezvous peers. Advertisements are indexed by values of so called index fields, which are selected elements of the advertisements themselves. An index entry is tagged with a lifetime value, i.e. the entries are leased by the publishers. In case the publisher disappears and the rendezvous does not notice that, the lifetime protects the index from holding long-expired entries. The discovered advertisements can be stored also in the discovering peer cache. Another timer defines the time such advertisement can be cached for.

The values of the lifetime and caching time are especially important given that the JXTA middleware does not define any mechanism for removing advertisements from remote caches. The peer that publishes the advertisement is able to remove it only from the local cache.

The Discovery Service handles only three types of advertisements, which are Peer Advertisements, Peer Group Advertisements and all other advertisements. There is no elegant way to make the middleware distinguish between its advertisements and user-defined ones. As a workaround, the framework prototype defines well-known unique index fields for its advertisements and queries the service using only those fields.

JXTA Discovery Service is not obliged to return all advertisements that match users' queries. Due to the scalability reasons, it is designed to return a subset of such advertisements. Given that, one would expect that the interface of the service would support queries of high granularity. Sadly, the only filter the interface allows is a text wildcard on one of index fields. Such functionality is sufficient for the framework prototype, but could result in problems in introspection-intensive applications.

Communication Services

The JXTA middleware provides two services which match most of the requirements of the framework prototype: Peer Resolver Service and Pipe Service. The former covers the Peer Resolver Protocol. Its idea is simple – communicating parties (services or applications) need to register unique "resolver hooks", which can be interpreted as communication port names for message multiplexing and demultiplexing purposes. Next, the communicating parties construct their messages, which are tagged with source and destination peer identifiers as well as with the resolver hooks. The messages are routed through the JXTA network and reach their target peers and services. The Resolver Service offers only best-effort message delivery, so in order to handle longer conversations, the communication parties need to implement reliability-related mechanisms themselves.

The Pipe Service implements the concept of a virtual communication channel, in JXTA terminology called a pipe, which may support reliable delivery as well as packet sequencing, and even encryption of transmitted data. The service covers the Pipe Binding Protocol,

which is based upon the Peer Resolver Protocol, and used for pipe initialization. The service is able to construct either point-to-point or point-to-multipoint communication channels (the latter called propagate pipes).

In most cases framework internal protocols do not require long-lasting connections, because the information needed by the communication target is included in one message. Therefore, Resolver Service, which offers a simple messaging interface, is sufficient for their functioning. Only the Deployment, Component Generation and Code Repository services need to create reliable communication channels for storing the code or downloading the deployed module code to the target worker.

Rendezvous Service

Peer-to-peer networks constructed with JXTA are hybrid, i.e. some of their members play the role of "super peers". The role of rendezvous peers, which form a subset of super peers is to maintain advertisement indexes, support regular peers in searching for resources and handle message broadcasting. A regular JXTA enabled peer typically creates a connection to a rendezvous at startup. The connection is created, maintained and destroyed by the Rendezvous Service. The framework prototype does not benefit from Rendezvous Service in any direct way.

Membership Service

Membership Service is a basic security-related service. Its task is to verify the authentication credentials each peer has to submit when joining a JXTA peer group. After successful verification the group member is given access to the services offered by peers that joined the same group. The scope of services provided by the framework prototype is also limited by group boundaries. The prototype uses the default implementation of Membership Service, which allows membership in a group to all peers that are able to instantiate the peer group services.

Modules and dynamic code loading

Module is one of key JXTA concepts. In short, module is a runnable, JXTA compatible code that can be downloaded and started by any peer. Modules (just like other resources) are advertised. However, unlike ordinary resources, modules have three types of associated advertisements: module class advertisement, module specification advertisement and module implementation advertisement. Module class advertisement serves as a common label for modules that implement similar functionality. Module specification advertisement identifies module interface, and contains e.g. well-known Resolver Service hooks that may be used to send messages to the module. The role of module implementation advertisement is to carry a reference for the place the module code is stored and to denote the specification implemented by the code. The reference may be used to download the code if no local copy is available.

In order to load a module, JXTA peer needs to know its specification advertisement. Given the advertisement, JXTA runtime is capable of searching for an implementation and loading the code. However, JXTA code loader is not capable of loading modules using JXTA pipes. It inherits most of its functionality directly from the Java 2 class loader, which of course has no knowledge of the pipe concept. Therefore, the Deployment Service prototype had to implement a custom module loader.

Dynamic code loading is used in another way by all of the framework services implementations. For user convenience, all framework prototype services are as peer group services. However, JXTA requires the peer group services to be loaded on any peer that joins the group. In order not to instantiate services that would not be used, when joining groups, only service stubs are loaded. The service code is loaded by the stubs when needed.

Implications of JXTA on overlay network implementation details

Framework instances must be able to operate in changing environment and therefore cannot rely on any centralized registry of entities. In such case, the ability to discover the entities and their dependencies becomes especially important. Therefore, framework support for introspection is limited by the capabilities of the discovery service. Because the framework introduces a set of specific entities (e.g. message sources, sinks, overlay networks, etc.), in order to make the framework introspection-enabled it must augment the discovery service with mechanisms that would differentiate the entities from all the others.

The proposed framework introduces the notion of an overlay network advertisement, which describes the structure of the (already created) overlay (Fig. 34).

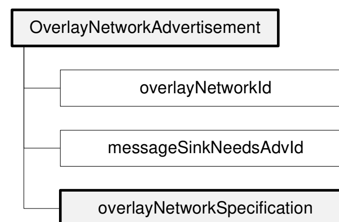


Fig. 34. Advertisement of an existing overlay network.

The overlay network advertisement consists of:

- an identifier of overlay network,
- an identifier of the message sink request that resulted in creating the network,
- the network specification the implementation satisfies.

The network is identified by an uniform resource name (URN) that is, by definition, “intended to serve as persistent, location-independent, resource identifier” [Moats 1997]. The URN contains a universally unique identifier (UUID) [Leach 2005]. The framework prototype reuses the format of JXTA’s module specification identifier to identify overlay networks. Moreover, two least significant hexadecimal digits of overlay network identifiers are by convention set to zero in order to facilitate easy creation of UUIDs for selected overlay network nodes. The main reason for creating UUIDs from locally unique addresses is to reuse the deployed components in other overlays. The UUIDs are created simply by substituting the two trailing zeros with component number, which must be locally unique. Note that such approach does not limit the number of processing nodes to 256 – it limits only the number of nodes that can be reused by external networks.

Overlay networks are created based on user requests published by message sinks. In case of a failure, the message sinks may re-issue their requests. Based on the request identifier, overlay network manager is able to repair a broken overlay instead of reinstantiating all its elements. Because the advertised networks are already created, the source code of message processing components need not to be carried by the overlay network advertisement, thus making the document shorter.

4.1.2. XMLBeans toolkit

JXTA messages are implemented as structured XML documents. However, the API used to create and modify message contents is far from convenient. Moreover the fields constructed with JXTA API introduce about 50 bytes of serialization overhead each. Therefore even the official JXTA Programmers Guide advises against using many short fields.

XMLBeans is a convenient toolkit for manipulating XML documents. Given an XML schema, generates a series of classes that bind the document to the Java 2 code. Moreover, the generated classes are capable of checking schema conformance of created documents, providing for better error detection. As an alternative, Sun's Java API for XML Binding (JAXB) could be considered and was indeed used in an early version of the framework prototype. However, due to its better coverage of XML Schema language, XMLBeans was finally chosen.

4.1.3. IODT Minerva

Minerva is an ontology storage, inference and querying system implemented as a part of the IBM Integrated Ontology Development Toolkit (IODT)[IBM 2006]. It is capable of processing ontologies written either as RDF or OWL documents and executing SPARQL queries either using its own or third-party reasoner (e.g. Pellet[Sirin 2007] or Racer[RacerPro 2007]). REMP prototype uses Minerva as the base for a matching service worker (Fig. 35).

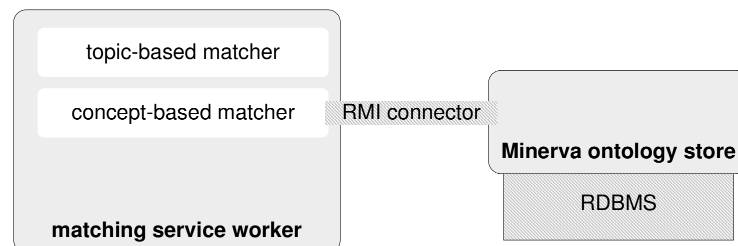


Fig. 35. Prototype matching service worker capable of executing concept-based queries.

The worker polls the JXTA-provided Discovery Service in order to obtain advertisements of message sources. If the message source advertisement contains an OWL ontology that describes the meaning of the messages the source can provide, appropriate fact, saying "source <X> is capable of publishing message <M>" is stored along with the published ontology. The prototype does not introduce any inter-worker synchronization mechanisms, save that all workers use the same Discovery Service.

Matching service workers publish their configuration advertisements. In order to let the proxies know, which worker is capable of executing queries expressed in particular language, the list of served languages is the main part of the published configuration.

The consumers using the REMP prototype instances are able to write parts of their queries regarding raw message sources in SPARQL. The queries take the form "which source is capable of publishing messages that meet the condition <C>". Once the proxy part of matching service instantiated by overlay network manager notices the sub-query expressed in SPARQL, it forwards the query to the appropriate worker.

4.2. A minimal configuration of framework prototype instance

In order to set up a fully functional framework prototype instance, one needs to run:

- at least one overlay network manager,

- at least one worker peer for:
    - matching service,
    - component generation service,
    - code repository service,
    - deployment service.

Because the prototype is implemented upon JXTA, which constructs hybrid peer-to-peer networks based on super-peers called rendezvous points, one rendezvous needs also to be provided. The minimal configuration is depicted in Fig. 36.

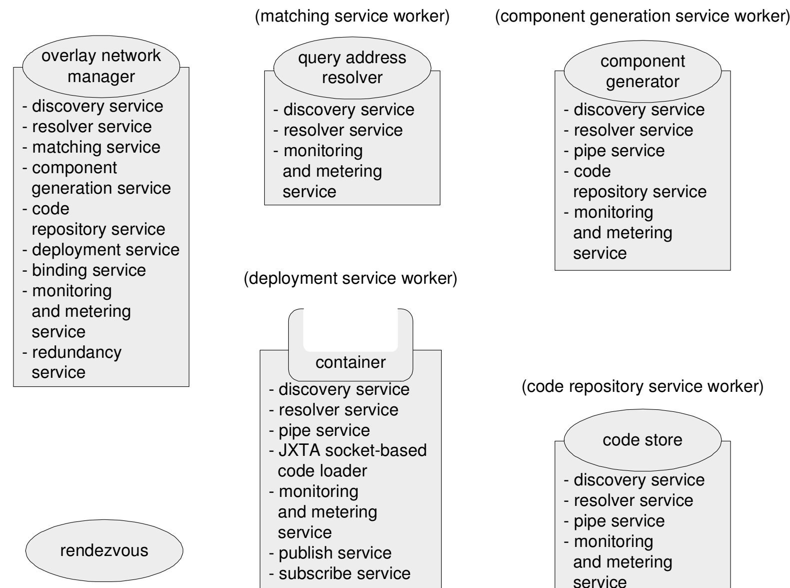


Fig. 36. A minimum set of peers constituting internals of a fully functional framework instance.

In order to perform their functions, peers of particular types are expected to use various services provided either by JXTA, or by the framework (see Fig. 33).

The following sections present the way the main use cases are implemented upon the prototype.

4.3. Implementation of key use cases

This section is dedicated to describe the cooperation of services pointed out in Fig. 33 when executing the basic use cases. Section 4.3.1 refers to the producer-initiated use cases, while 4.3.2 discuss the consumer-initiated one.

4.3.1. Use cases: "produce messages", "get available message streams"

Message producers are the sources of message streams that undergo processing inside a framework instance. In order not to make the message producers deal with low-level issues related to the framework internal protocols, an interface service (Publish Service) is implemented. The service wraps the internal protocols and provides an intuitive interface to the end user application (Fig. 37). The chosen approach makes it possible to reuse JXTA discovery service for retrieval of available message sources descriptions. The following paragraphs describe the Publish Service in more detail.

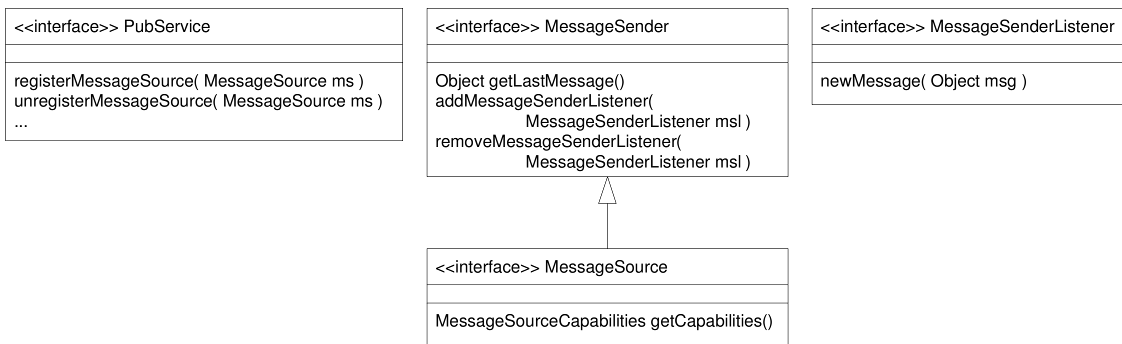


Fig. 37. Message producer application programmer interface.

Publish Service

The Publish Service is instantiated on any message producer as well as on the Deployment Service workers (i.e. containers, described later). The interaction between message producer and Publish Service is depicted in Fig. 38.

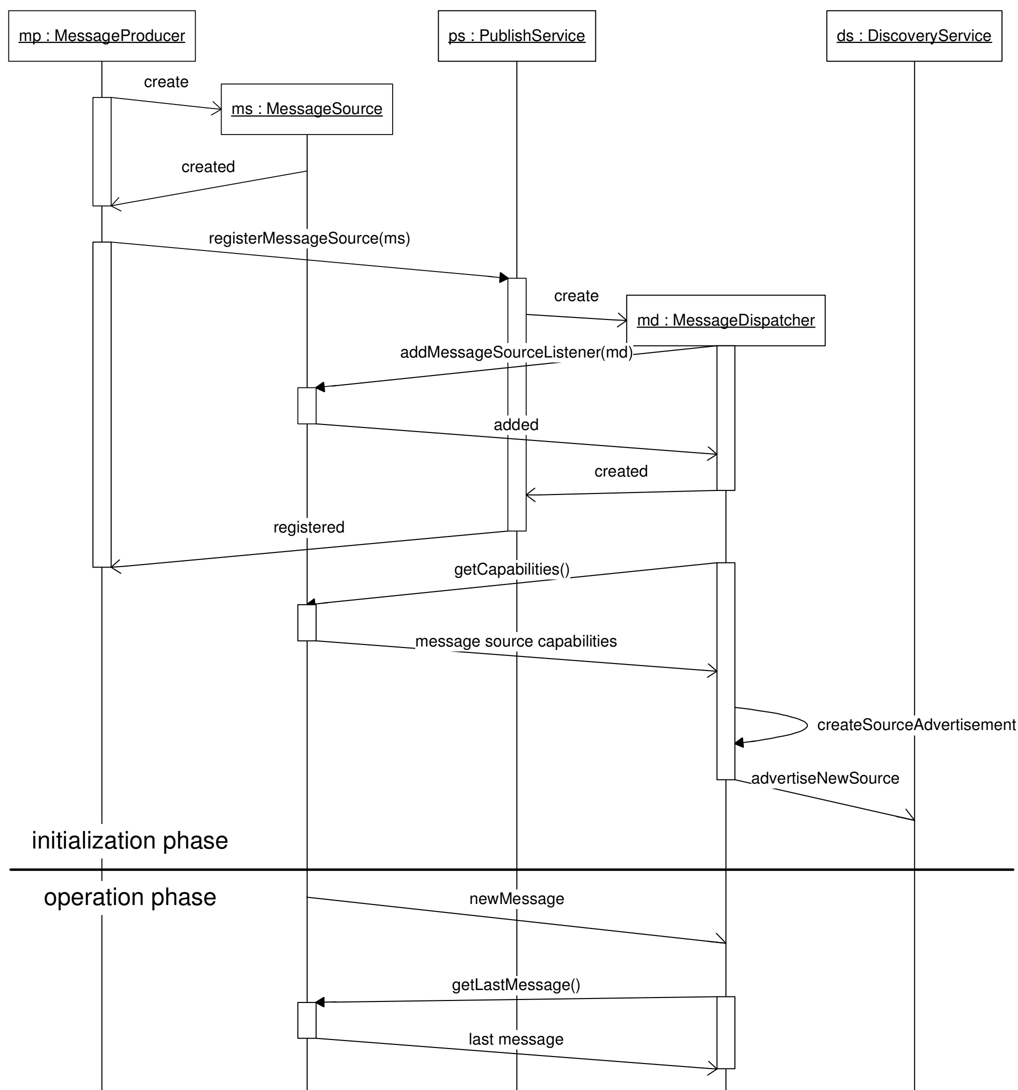


Fig. 38. Sequence diagram for interaction between a message producer and Publish Service.

From the user's point of view, Publish Service provides means for message source registration. During message source registration a handler thread, called message dispatcher, is created for each registered source. The dispatcher registers itself in the source as a listener for new messages and obtains the user-defined description of the message

source's capabilities, including descriptions of semantics and syntax of messages and proposed sets of QoS parameters for communication channels that will be created. The contents of the document obtained from the message source are depicted in Fig. 39.

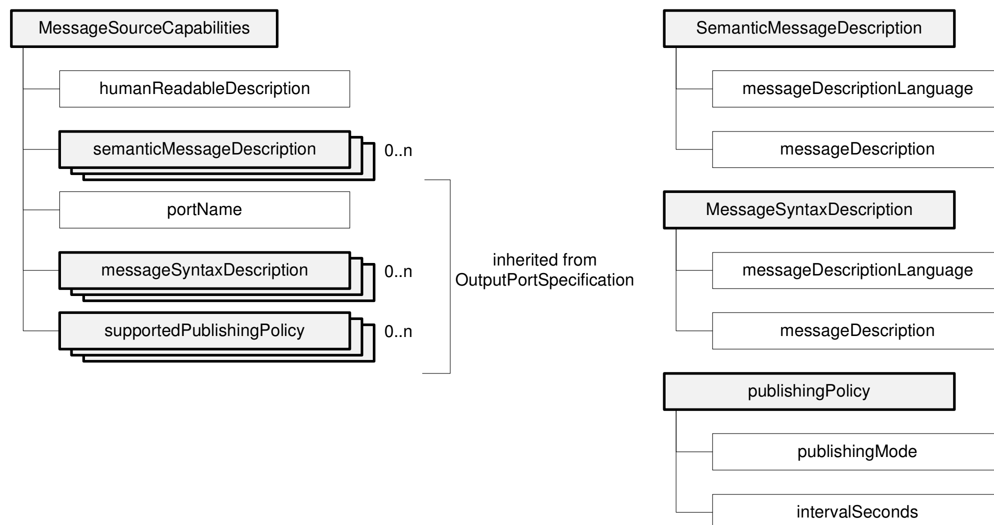


Fig. 39. Contents of message source capabilities description.

After the capabilities description is obtained, a message source advertisement is generated and published using the Discovery Service. In this way, any interested party can discover the capabilities of particular message source. The advertisements are of particular interest for Matching Service workers (described later), which perform matching between sub-queries and source advertisements.

From the framework's point of view, instances of Publish Service are responsible not only for publishing the advertisements of registered message sources, but also for creating communication endpoints on Binding Service's demand (the interaction is described later in this chapter). Therefore, message source advertisements carry not only the description of sources' capabilities, but also meta-information needed to ensure proper interaction between framework entities. A crucial part of this meta-information is the identifier (logical address) of the peer the advertised message source is running on. The identifier is used to contact the respective Publish Service in order to create a communication channel by Binding Service instances.

After at least one outbound communication endpoint is created, the message dispatcher becomes responsible for obtaining messages from the registered source and forwarding them to the communication channels. A message can be produced either on the Publish Service demand, or when the message producer decides to publish one.

Reuse of the JXTA Discovery Service

Message streams available at registered sources are advertised using standard JXTA mechanisms. A custom advertisement type, called MessageSourceCapabilitiesAdvertisement was created for that purpose. In order for the type to be easily distinguishable from other custom advertisements, a separate index in JXTA advertisement registry is created. Peers that want to know the sources registered in the system, should query the discovery service using "CapabilitiesId" as the index field parameter. An example advertisement storing and retrieval scenario is depicted in Fig. 40.

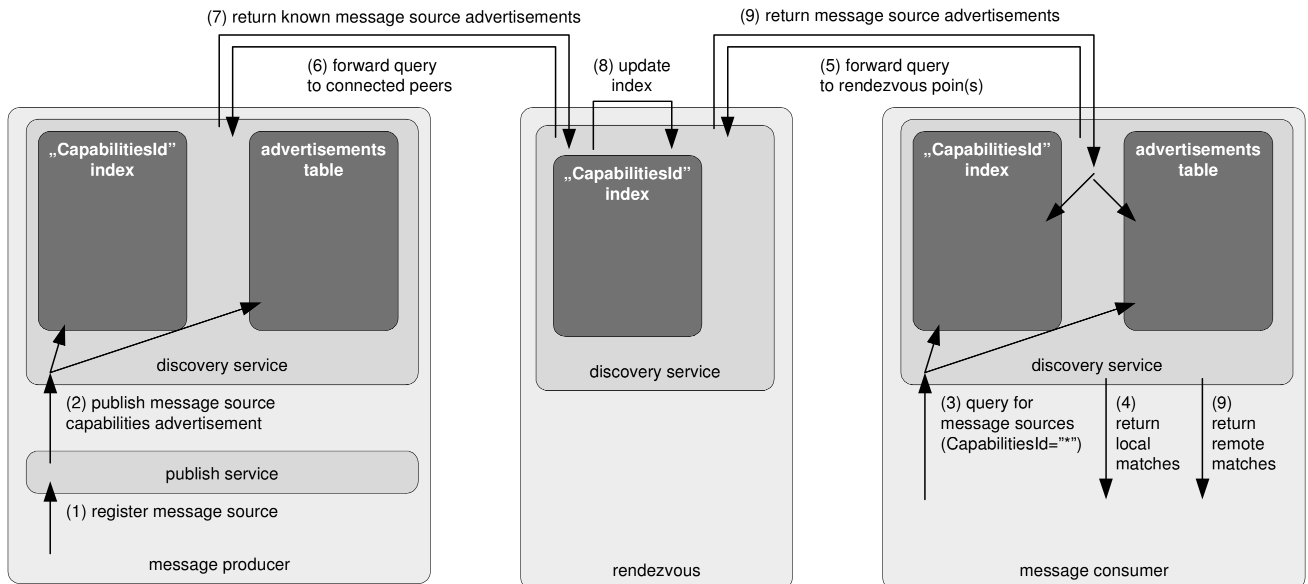


Fig. 40. Message source advertisement publication and retrieval (example).

As already described, a message producer register message source objects in its local instances of publish service. During the registration, a message source capabilities advertisement is registered in respective discovery service. In fact, the advertisement is stored in local cache (advertisements table) and entries in respective indexes are created. Optionally, the index entries may be forwarded to rendezvous peers known by the producer. A client, typically a message consumer, that wants to discover message sources present in the system registers its query in its local instance of discovery service. The service provides information both about locally stored advertisements (which do not have to be locally originated), and remote advertisements, obtained by forwarding the query to rendezvous points.

4.3.2. Use case: "create a message processing overlay network"

Message processing systems rely on message consumers to initiate the distribution of messages by registering their interests, i.e. by issuing queries for messages. The needs expressed by the consumers carry not only the queries that are used to identify the set of possible message sources, but also additional information regarding e.g. the rules for filtering and merging message streams or quality of service requirements. In fact, they specify the overlay networks to be created by the message processing systems.

The conversation between message consumers and a framework instance was outlined in section 3.2.1. The conversation details are wrapped by Subscribe Service, which serves as an interface between user application and system internals (Fig. 41).

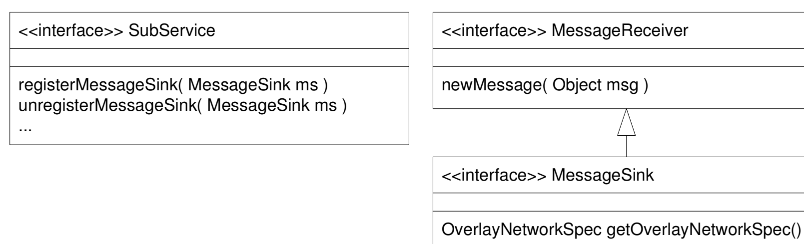


Fig. 41. Message consumer application programmer interface.

The following paragraphs describe the interactions that implement the most important parts of the overlay network creation process initiated by a message consumer. Firstly, they describe the calls that are performed on client's machine, when the message consumer invokes a Subscribe Service instance. Then, the most important elements of overlay network specifications and their handling are discussed.

Subscribe Service

The Subscribe Service is instantiated on each of message consumers as well as on the deployment service workers. From the user's point of view, the service provides means for registering needs of message consumers, which are represented by message sink objects. The interaction between a message sink and Subscribe Service is presented in Fig. 42.

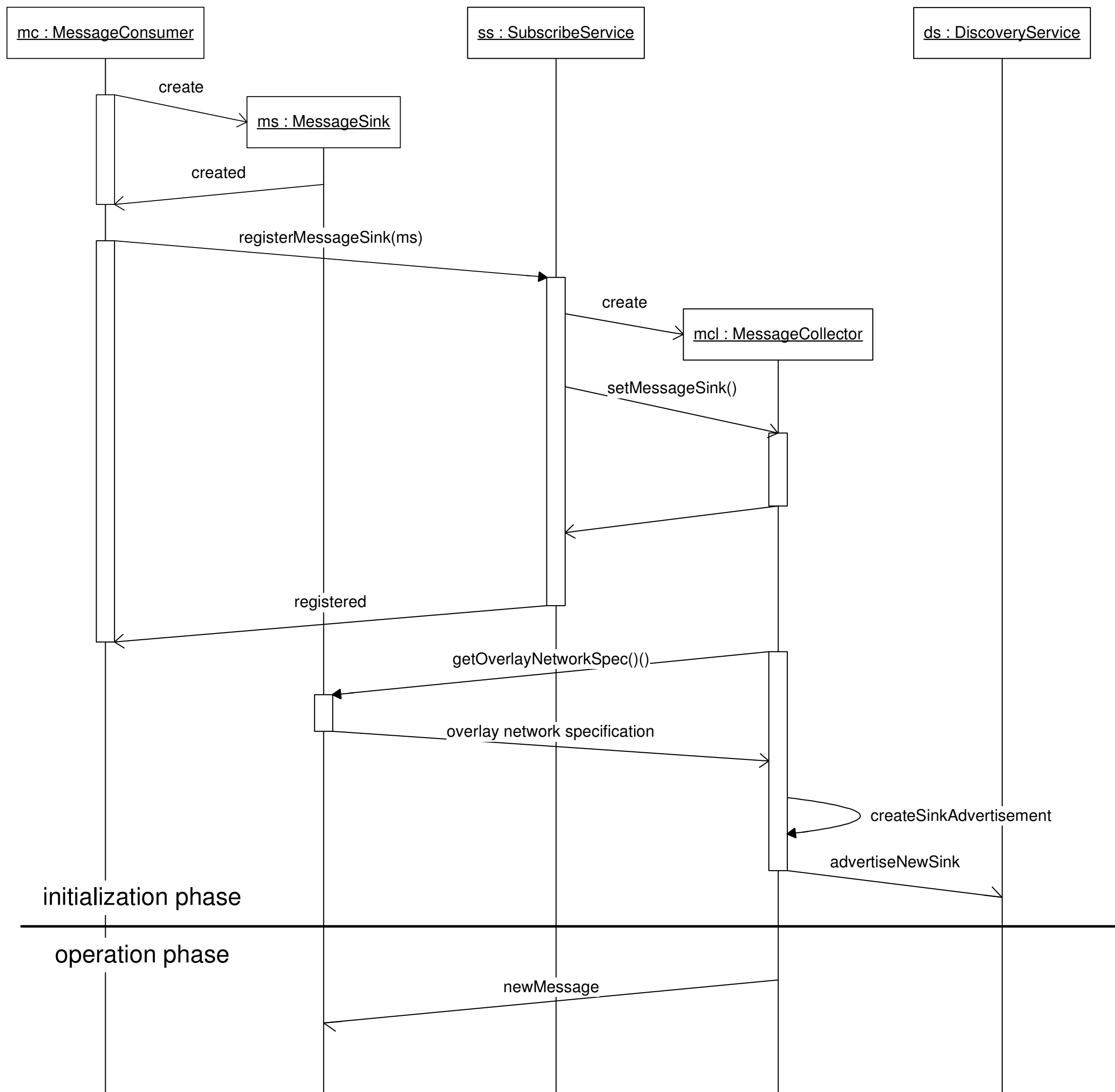


Fig. 42. Sequence diagram for interaction between a message consumer and Subscribe Service.

During message sink registration, the Subscribe Service creates a message collector, which is handler thread for the registered message sink. The collector registers itself in the sink in order to provide it with its (the collector's) reference. Depending on the message delivery scenario chosen by the sink, it may either be notified asynchronously whenever a new message arrives, or poll the collector for arrived messages.

After registering itself as a listener, the message collector obtains the user-defined specification of the overlay network to be created. The description is then encapsulated in a message sink needs advertisement, which is published using the Discovery Service. By querying the service an overlay network manager is able to discover message sinks and process their requests. Note that neither the message sink, nor the message collector needs to know the manager reference, i.e. the request is addressed to “any overlay network manager”.

From the framework internal entities’ point of view, the instances of Subscribe Service are responsible not only for encapsulating and publishing of message sink needs advertisements, but also for calling local instances of JXTA Pipe Service to create inbound endpoints of communication channels on remote Binding Service’s demand. To support that, the message sink needs advertisements carry not only overlay network specifications, but also meta-information needed to ensure proper interaction between framework entities, including the reference of the peer that hosts the message sink of interest. The reference is used by Binding Service instances to properly address a request for creation of a new communication channel (the interaction is described later in this chapter). Other services, such as the Monitoring Service also may benefit from that information. After an inbound endpoint is created, the message collector becomes responsible for notifying the registered message source about newly received messages.

Overlay network specification processing overview

The overlay networks are constructed in accordance with the specification of their contents issued by a message sink. A sink can have many input ports that are linked to various nodes (see Fig. 43). Such approach limits the spectrum of possible message distribution graphs to single-sink ones. It would be possible to extend the specification to include multiple sinks, but for two reasons it does not seem appropriate. First, the sinks represent peers external to the framework instance, i.e. lying outside the its management domain. Therefore, a framework instance cannot create communication channels to the sink in response to another sink’s query. Second, introducing support for multiple sinks would complicate the graph specification, while not introducing any new possibilities to the framework itself. Multiple sinks can be supplied with the same messages by using one query containing full specification of the overlay network and several simple specifications of external links referring to the UUIDs of reused nodes, created in accordance with the rules described in section 4.1.1.

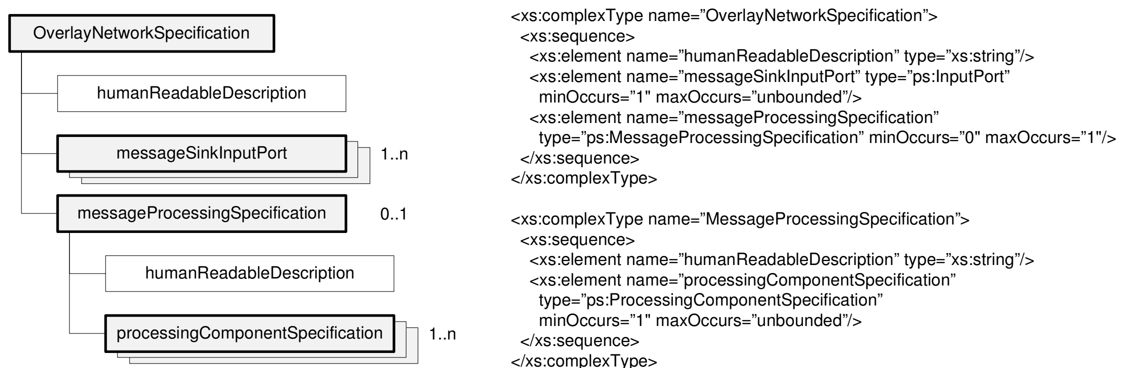


Fig. 43. Contents of the overlay network specification.

As discussed in section 3.2.1, the contents of an overlay network specification are interpreted by an overlay network manager, which, in turn forwards respective parts of the specification to appropriate services (see Fig. 19). The following subsections discuss the

most important parts of the processing – resolution of the concept-based addresses into source references, message processing component creation, deployment and binding.

Concept-based address resolution

The task of finding message sources that match parts of the overlay network specifications is assigned to the Matching Service. The service is implemented using a proxy-worker pattern.

The proxy selects the worker based on the query language it serves. Because the message sinks are allowed to specify their queries in various query languages, it is expected that multiple Matching Service workers serving multiple languages may be present in a working configuration. If the proxy finds more than one worker capable of interpreting the query language, it may use additional criteria, e.g. usage statistics obtained by calling the Monitoring Service to obtain usage statistics for each worker.

The Matching Service workers are responsible for collecting advertisements of message sources capabilities and resolving queries issued by proxies against the collected database. The framework prototype implements two options for Matching Service workers operation: single-call and continuous. In single-call mode, the worker resolves the query against database only once, and sends the results to the query issuer. In continuous mode, the query is evaluated not only against the database, but also against any new message source capabilities advertisement that is discovered (Fig. 44). In the continuous mode, the worker calls back the proxy that issued the request for sources any time a new source that matches the query is found. If the proxy instance is located at an overlay network manager, that tracks the new deployment possibilities such notification may cause migration and re-linking of a part of the overlay network.

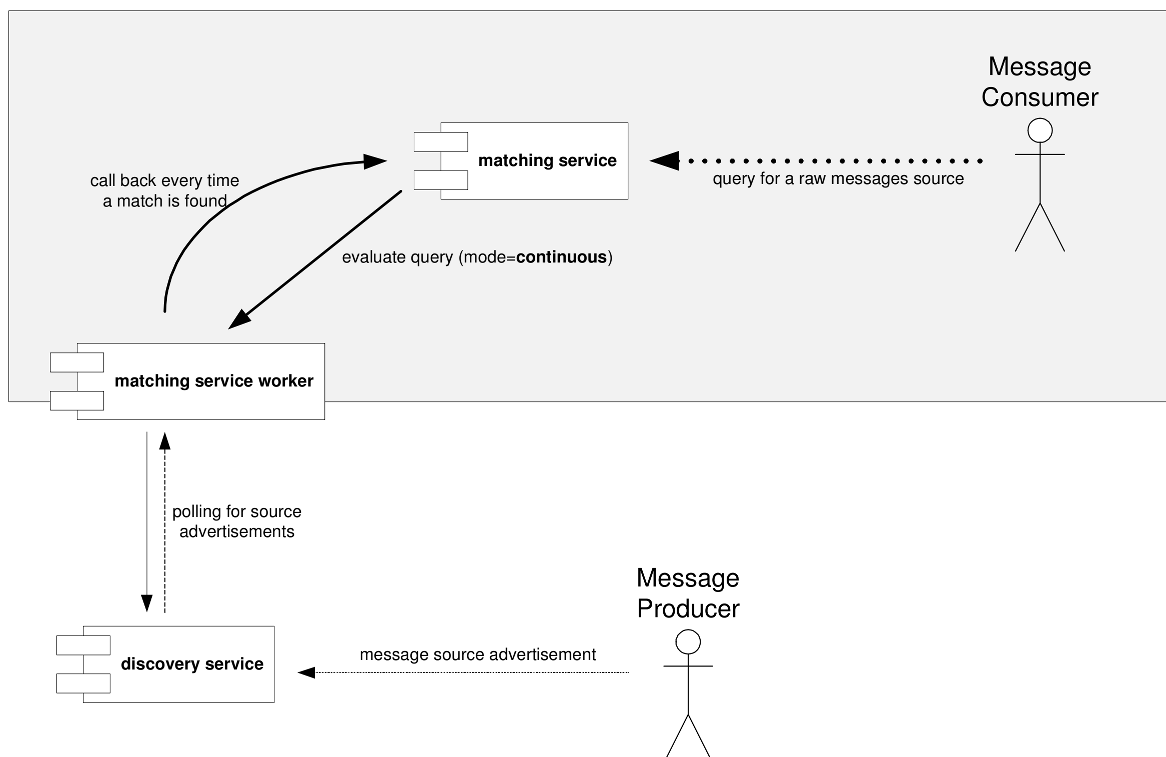


Fig. 44. Resolution of a sub-query regarding raw message sources.

In order to make framework customization easier, the prototype implementation provides a generic matching service worker that wraps the proxy-worker protocol from the

message processing system developer. Extensions to the stub must implement a Matcher interface, depicted in Fig. 45.

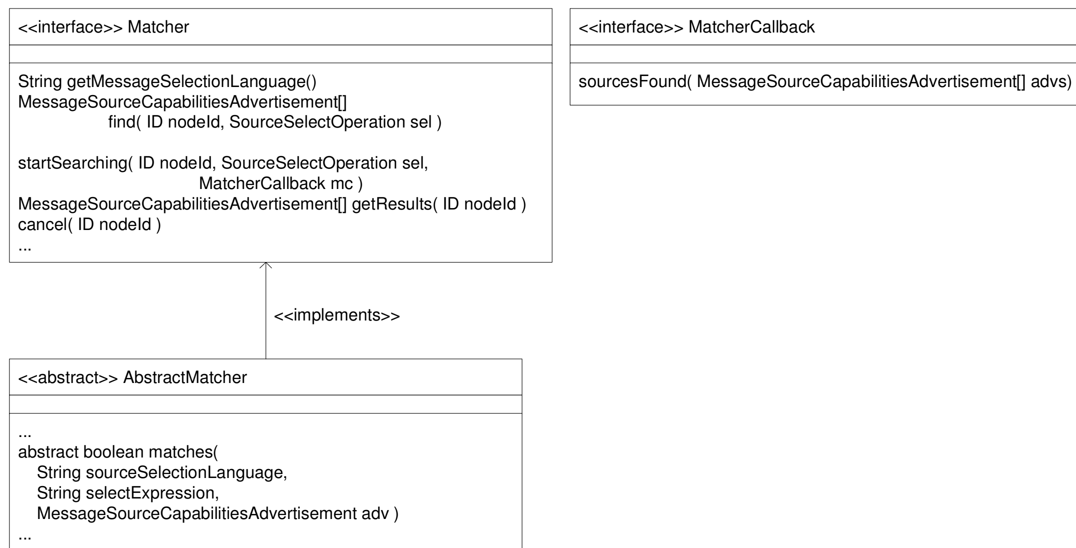


Fig. 45. Matching service worker's programmer interface.

Simple workers might extend a default implementation of the Matcher interface, which implements the details related to polling the discovery service, starting matcher tasks, etc., while leaving only one method (for checking a single advertisement against a single sub-query) abstract.

Message processing components creation

Message processing nodes are the basic elements of deployed overlay networks. A message consumer that issues a query has an option to specify the delegated functionality, i.e. is able to insert some code expressed in a language served by the message processing system it is registered with. However, because the internals of the message processing system should be as opaque as possible to its clients, the message consumer is not required to program a fully functional component. Rather, it is expected to provide fragments of such component's code and let the component generation service extend and compile them (Fig. 46). The deployed and instantiated components are interlinked with other overlay network entities. Therefore, the process of generating of components requires the message consumer also to declare what input and output ports should a particular component be equipped with. The contents of the processing component specification are depicted in Fig. 47.

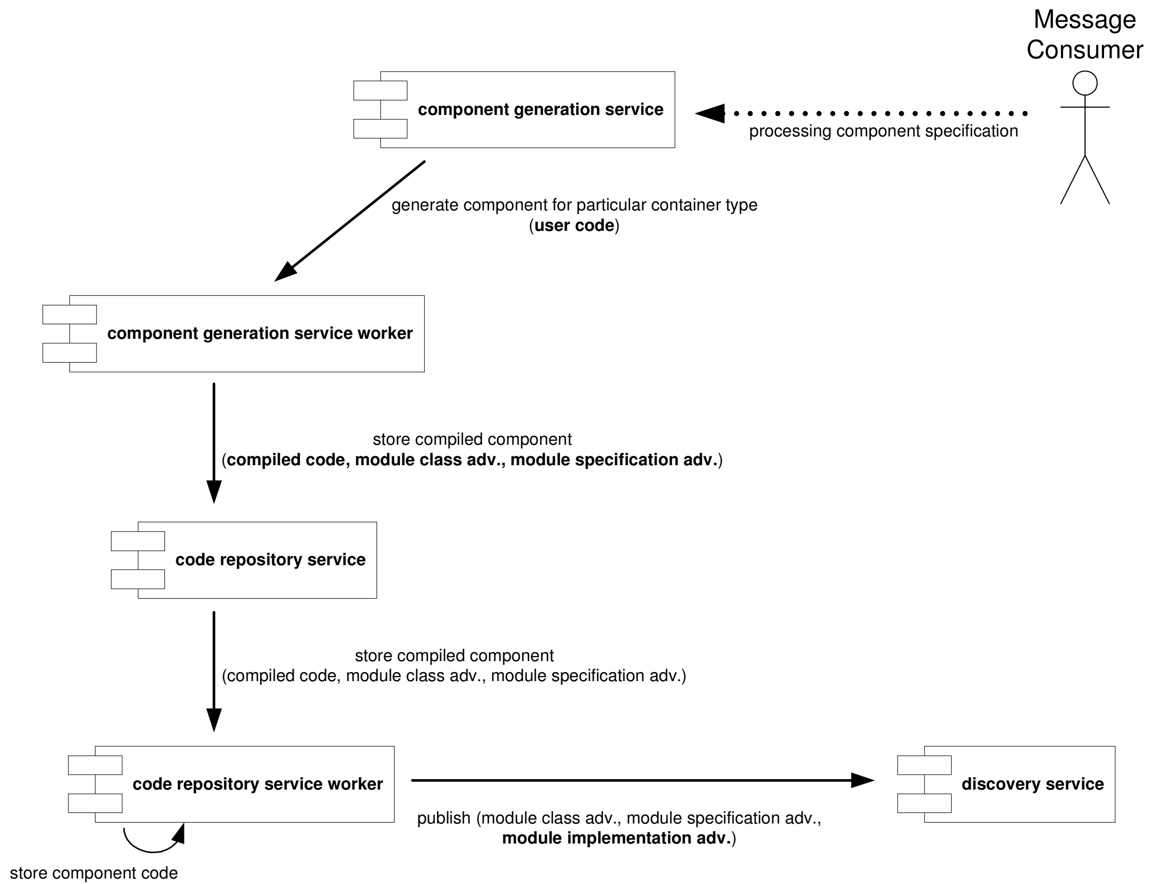


Fig. 46. Generation of a deployable component based on processing component specification provided by user (simplified).

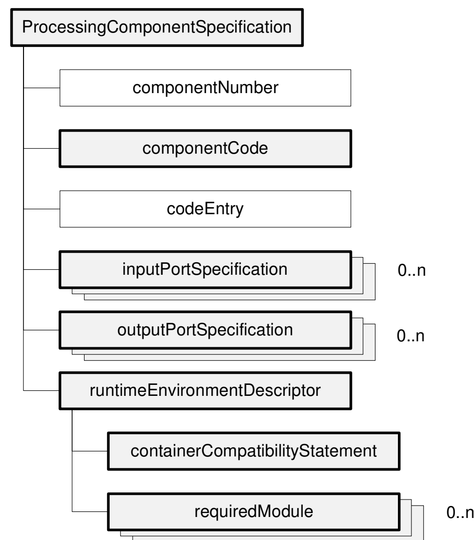


Fig. 47. Contents of the processing component specification.

The component number is a locally significant positive integer identifier of the component. It is used mainly by the Binding Service that interlinks the overlay network entities. The identifiers are unique only locally in the overlay – message sink that is responsible for specifying its needs, is also responsible for assuring the local uniqueness. If needed, the UUIDs for components can be constructed by simply adding the value of the locally unique identifier to the overlay network UUID.

The element called `componentCode` is used either when a user delegates the task of generating the component executable code to the framework, or reuses a previously stored component. In the first case it consists of the source code language specification which is used by the framework to find appropriate component compiler (component generation service worker) and of the source code itself. In the second case, it simply holds the module specification identifier of the previously stored component. The `codeEntry` element is used by the Deployment Service that starts the compiled component in a container and denotes the starting point for component execution. The exact value of the field depends on the component code, e.g. for the Java-based component it is the name of the main component class.

The networks created by the framework are composed of components deployed on participating peers. The communication channels are created between containers that map their endpoints to input and output ports of respective components. To provide message forwarding, the framework relies on JXTA Pipe service. Three basic types of pipes provided by any JXTA implementation are:

- unreliable point-to-point pipe,
- unreliable propagate pipe,
- secure and reliable point-to-point pipe.

Thanks to JXTA community members work, there are more options for communication channels. These include bi-directional pipes and propagate pipes based on reliable multicast [Bobowiec 2006]. The properties of a particular channel are specified along with the input port specification details (Fig. 48).

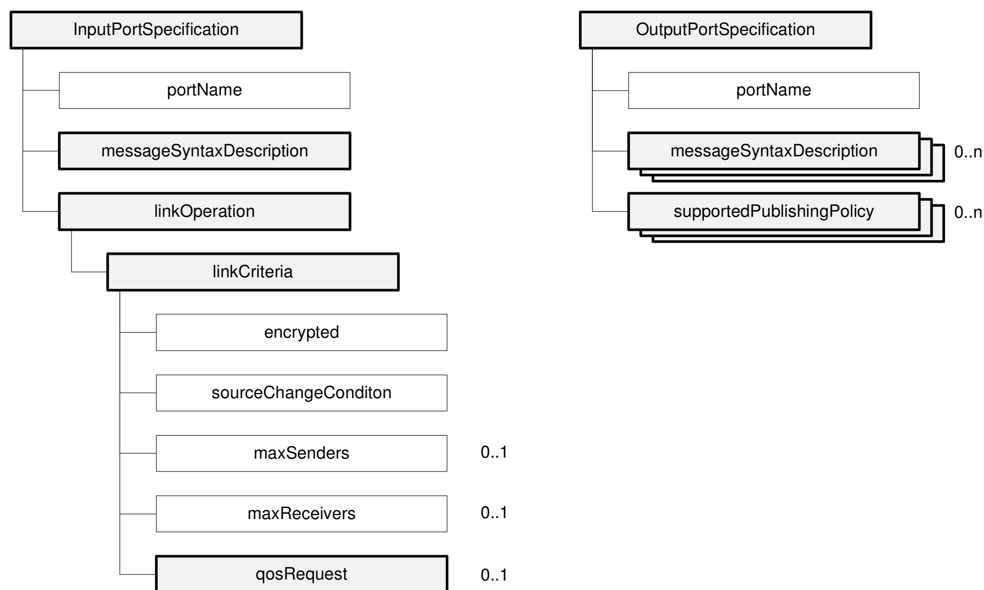


Fig. 48. Contents of the input and output port specifications.

The framework reuses the channels provided by the middleware. The specification of the receiving end of any channel (input port specification) contains a set of hints that are used by the framework to instruct the underlying communication service about the type of channel to be created. The selection of the channel type is based on the maximum multiplicity of senders, receivers and requirements for channel encryption included in the specification of criteria the link must or should meet (`linkCriteria`).

The framework introduces simple, hop-by-hop QoS mechanisms. The QoS options (`QoSRequest`) include selection of channel operation mode from: ON_CHANGE, ON_DEMAND and PERIODIC as well as specification of the interval between subsequent

messages (meaningful only for periodic message delivery). The requested QoS parameters are compared to the QoS offers specified by message sources and if a match is found, a channel is created or reused. The prototype implements the “push” model for end-to-end message delivery and periodic polling for raw messages.

Because the message processing nodes are expected to be run remotely, on an unknown peer, means for restricting their runtime environment (e.g. for specifying required libraries) must also be provided. That task is assigned to runtime environment descriptors carried along with the component specifications. The statements consist of a mandatory part that identifies the container type (e.g. JXTA2.4/Java2) and an optional list of modules required by the component.

Component Generation Service

The goal of the Component Generation Service is to wrap and compile the code of intermediate nodes that implement the functionality delegated from message sinks (see Fig. 47). The service is designed similarly to the Matching Service – in accordance with the same proxy-worker pattern. Proxies – local to the caller – are responsible for selection of workers, submission of node generation jobs and collection of the results. The primary criteria for worker selection include the high-level programming language the worker is able to compile as well as the compilation environment (libraries, etc.). A proxy may also call the Monitoring Service to obtain reports about specific workers and choose e.g. the one that was rarely used recently.

The workers are responsible for processing the requests issued by a proxy by generating additional source code needed to instantiate the generated processing (intermediary) node on a deployment service worker. The generated code contains a set of wrappers that isolate the processing node from underlying middleware. From the user’s point of view the wrappers are responsible mainly for delivering messages to and receiving them from the processing code (Fig. 49). Although the framework model recognizes only the need for a single wrapper, the prototype implementation differentiates between wrappers designed for message delivery and a single wrapper for instantiating and destroying the processing component. Input and output port wrappers are created along with the specification of input and output ports that is included in the processing component specifications. The prototype implements two kinds of output port wrappers: simple wrapper for overlay-network internal connections and a more featured one for output ports that should be advertised as message sources. In the future, wrappers related to input or output ports may also expose interfaces e.g. for message interceptors. The code wrapper serves as an intermediate between the input port wrappers, container and the processing code.

Fig. 49. Wrappers created by the framework’s prototype component generation service worker.

The prototype requires the user to express the processing code as a Java class. By introducing the wrappers, the requirements regarding the class were reduced to implementing a simple two-method interface (Listing 2). The first method (`startProcessing`) is called at component startup and is used to provide the component with a reference to its wrapper and its module implementation advertisement,

which may carry initial configuration and context information for the component. The second method (`newMessage`) is called by the wrapper whenever a new message is received.

```
public interface Component {
  public void startProcessing(
    ComponentWrapper wrapper,
    ModuleImplemetationAdvertisement adv )
    throws ComponentStartupException;
  public void newMessage( String inputPortName, Object message );
}
```

Listing 2. Java interface to be implemented by processing nodes compatible with the framework prototype.

The prototype implementation of ComponentWrapper offers its user three methods (Listing 3). First of them (`publishMessage`) provides the processing code with a way to publish a newly produced message. Second (`getLastMessage`) retrieves last message that was buffered by a respective input port wrapper. The third method (`deliverMessage`) is used mainly by the input port wrappers to deliver messages to the processing code, but might be also used by the code as a loopback call i.e. to send a message to itself.

```
public interface ComponentWrapper {
  public void publishMessage( String outputPortName, Object message )
    throws NoSuchPortException;
  public Object getLastMessage( String inputPortName )
    throws NoSuchPortException;

  public void deliverMessage( String inputPortName, Object message )
    throws noSuchPortException;
}
```

Listing 3. Prototype component wrapper interface.

The `Component` and `ComponentWrapper` interfaces are too simple to provide full functionality required by the framework model. For example, the prototype `ComponentWrapper` does not offer any method for using Redundancy and Checkpointing services and the `Component` interface does not yet define any “shutdown” method that could be used by component migration services. Clearly, the interfaces are subjects to extensions in more expressive implementations of the framework model.

The interfaces required to be implemented by user classes may vary, because framework instances may use various containers. The containers might offer various component execution environments, both in terms of offered services and requirements regarding the compiled code. If a particular instance offers various execution platforms, it is important to assure that each kind of containers is accompanied with appropriate component generator. Although it is technically possible to make the framework user pre-compile whole components and store their code using the Code Repository Service, for most of the users it is obviously inconvenient.

Code Repository Service

The goal of the Code Repository Service is to keep the compiled code of modules that may be instantiated and executed using the Deployment Service (the interaction is described later in this chapter). The service is designed in accordance with the proxy-worker pattern.

Proxies local to the caller gather configuration advertisements of workers and after being called, choose some of them to store module code. If asked, the workers not only store the code, but also advertise its presence, so that interested parties may discover the stored code (Fig. 50).

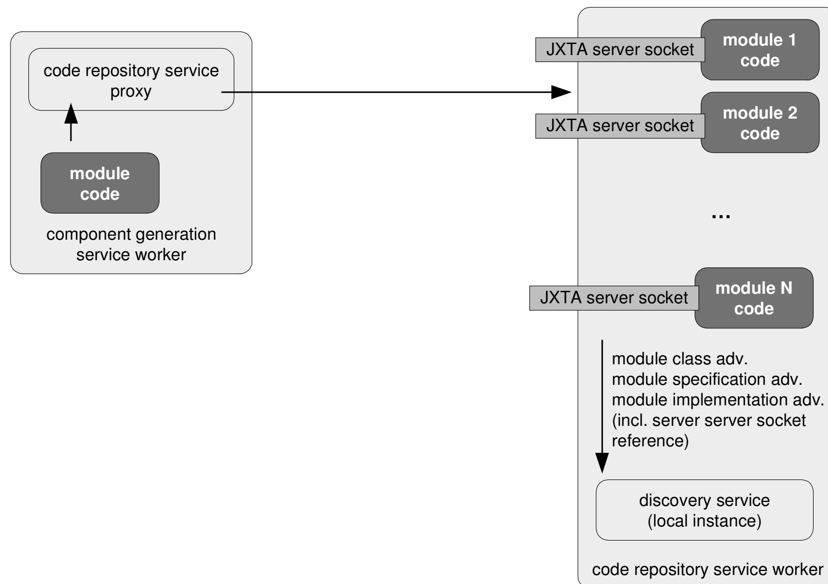


Fig. 50. Module storage in reference code repository service worker.

The workers provided by the prototype implementation allow downloading the stored code through JXTA sockets. For each stored module, a new JXTA server socket is created and its reference is placed in respective module implementation advertisement. A peer interested in downloading and instantiating the code, contacts the worker directly through a JXTA client socket.

The compiled code should be stored in at least two locations in order to facilitate single worker disconnection. The number of copies as well as the algorithm of choosing workers are implemented by the proxy part. The proxy may ask the Monitoring Service for usage statistics for particular worker, and choose the worker e.g. by its uptime.

Message processing components deployment

The process of deploying previously created message processing components is initiated by an overlay network manager after all needed message sources are found and all processing components are generated, compiled and stored in code repository service workers (see Listing 1). In order to make the task of deploying a component simpler, the deployment service wraps all the communication details of downloading compiled code from code repository service workers to the chosen containers.

Deployment Service

The goal of the Deployment Service is to allow dynamic placement of mobile code. As in the case of many other services, it is constructed according to the proxy-worker pattern. The proxies local to the caller gather their workers' (containers) configuration advertisements, and present the caller with a list of deployment options for a given component. Technically, a "deployment option" consists of the identifier of the JXTA module that represents the component to be deployed and the configuration advertisements of compatible containers. The configuration is already tested against the requirements regarding the execution environment presented by the module specification advertisement.

The caller then may choose a deployment option it wishes to execute based on any criteria it wants and call the proxy again to execute the option.

Once a deployment option is chosen, the proxy contacts the selected worker and instructs it to search for the code of particular module using the Code Repository Service. The worker downloads the module code, instantiates and executes it and signals the result to the calling proxy.

JXTA reference implementation offers a mechanism for automatic module loading, that is partially reused by the framework prototype. Given a module specification advertisement, any JXTA peer is able to search for appropriate module implementation advertisement, download the code and execute it. As described earlier in this section, code repository service workers expose the stored code through JXTA server sockets and advertise the socket references in respective module implementation advertisements. Sadly, JXTA socket references are not understood by peers based on JXTA reference implementation, because JXTA class loader, is only a simple extension of Java 2 class loader (in fact, there is even no standard syntax for JXTA sockets reference). However, because the workers are not expected to be used directly by user applications, that disadvantage does not yield severe consequences to the framework usability. For message processing systems developers' convenience, the prototype extends the functionality of default JXTA class loader in a transparent way. The augmented loader is able to search for respective module implementation advertisement, download and instantiate the modules stored on code repository service workers (Fig. 51). In case the module is executed in more than one instance, the loader uses its locally stored copy of the code.

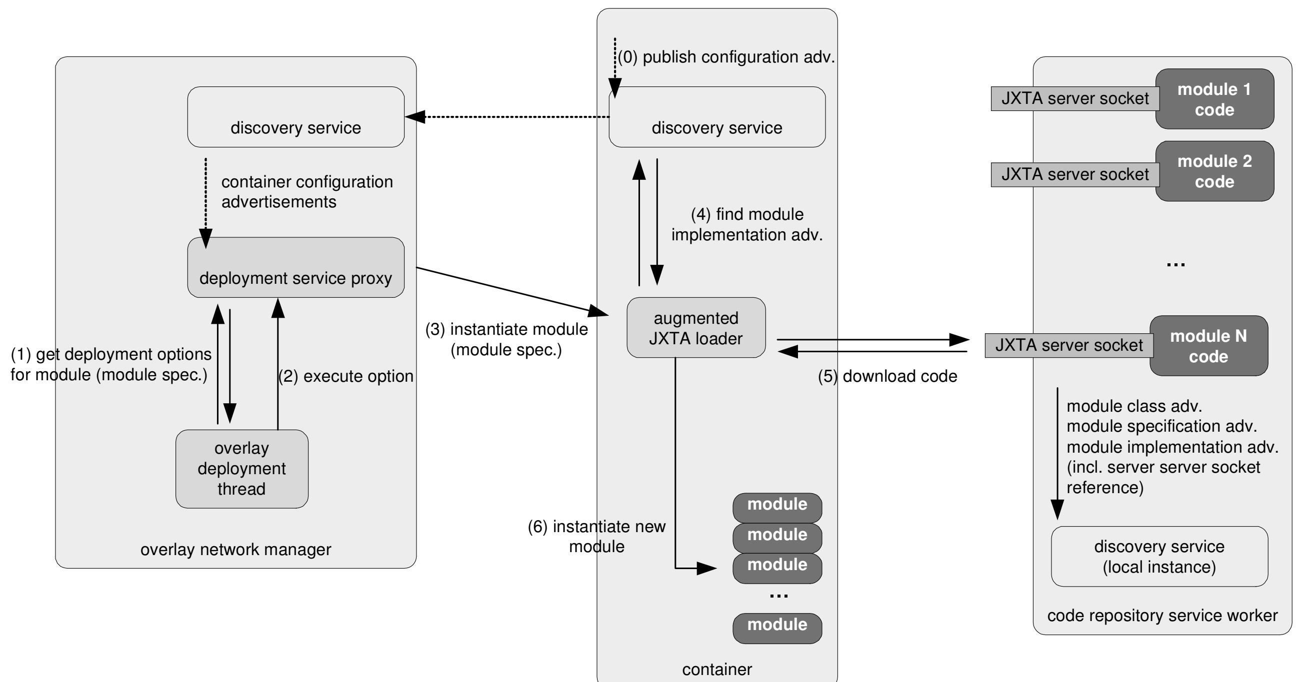


Fig. 51. Instantiation of a new module (simplified).

The downloaded modules are introspected in order to determine whether they implement the required Component interface and whether they belong to the privileged components set. Depending on the introspection result, they are instantiated with either already discussed `ComponentWrapper` or extended JXTA peer group interface that allows accessing all framework services (Listing 4).

```
public interface ExtPeerGroup extends PeerGroup {
  public PubService getPubService();
  public SubService getSubService();
  public MatchingService getMatchingService();
  public DeploymentService getDeploymentService();
  public BindingService getBindingService();
  public RedundancyService getRedundancyService();
  public CodeRepositoryService getModuleRepositoryService();
  public ComponentGenerationService getNodeGenerationService();
  public MeteringService getMeteringService();
}
```

Listing 4. Extended JXTA peer group interface.

The respective wrappers of processing components are registered in container-local instance of publish and subscribe services (Fig. 52), typically not as message sources and sinks, but as message senders and receivers (message senders and receivers are simpler objects, not publishing any advertisements).

In the case the processing component specification contained a `MessageSourceCapabilities` record instead of its simpler parent `OutputPortSpecification`, the generated wrapper implements a `MessageSource` interface and is registered as a message source presumably in order to provide the option of reusing intermediary computation results to entities external to the overlay network. Publish and subscribe services behavior with regard to message senders and receivers is similar to the case of message sources and sinks (see Fig. 38 and Fig. 42). The only difference is that no advertisements are published.

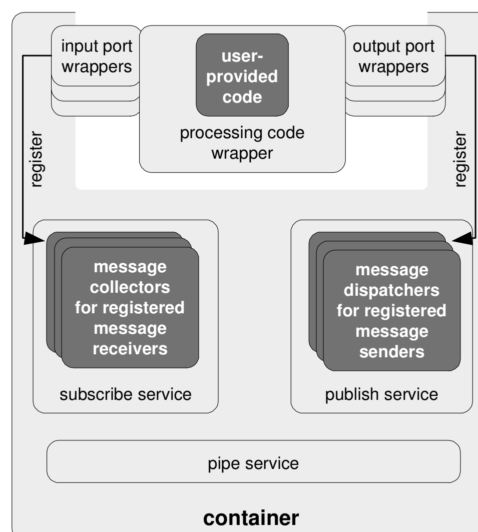


Fig. 52. Registration of component wrappers in container instances of publish and subscribe services.

After registration, the component deployment process finishes. Note that no communication channels are created until an instance of the binding service contacts respective instances of publish and subscribe services.

Binding the deployed message processing components with communication channels

In order to start message processing, the deployed and initialized components of an overlay network need to be interlinked with communication channels. Although channels of

multiple kinds are supported by the reused JXTA middleware, processing component specification does not contain any “component type” field. Rather, it contains a set of hints that guide the choice performed by overlay network manager. This subsection describes briefly the communication channel specific parts of overlay network specification as well as the functionality of the binding service.

Specification of requested outbound ports

The `LinkOperation` element type of the input port specification (Fig. 48) specifies the sending side to be connected to the described port. The type is actually abstract and extended by three other types:

- `SourceLinkOperation`, that carries sub-queries regarding raw message sources (to be passed on to the Matching Service),
- `InternalLinkOperation`, that describes the links between overlay network components, and
- `ExternalLinkOperation`, that describes the links to components of other overlay network (by using their UUIDs).

Moreover the `SourceLinkOperation` type is further extended by `SourceSelectOperation`. The contents of the extending types are depicted in Fig. 53.

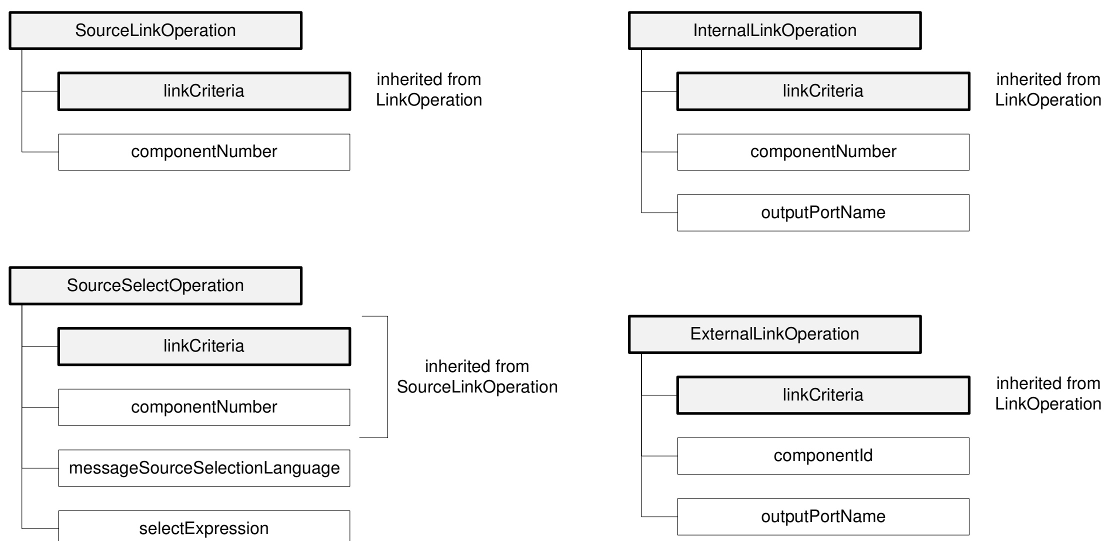


Fig. 53. Options for selecting sending ends of the inbound communication channels.

The presence of the `SourceSelectOperation` in any component specification makes the framework use its Matching Service as well as the underlying Discovery Service to find an appropriate message source. In order to find a source, the Matching Service needs to know the name of the query language (`messageSourceSelectionLanguage`) as well as the query for sources expressed in the `selectExpression` element.

Should the same `SourceSelectOperation` occur more than once in the same overlay network specification, the user is allowed to use its simpler form, namely the `SourceLinkOperation`, which carries the component number that must be equal to the one specified in the full query. In case of the simpler form, the framework prototype does not submit additional requests to the Matching Service, but reuses the results of former query.

The processing of `InternalLinkOperation` and `ExternalLinkOperation` elements is similar. The `InternalLinkOperation` carries a locally significant number

that identifies the component that is expected to produce messages as well as the name of an output port of the component. The actions taken by the framework are restricted to binding the components together. The `ExternalLinkOperation` is defined in a way syntactically similar to the `InternalLinkOperation`. The difference is that elements of this type carry a globally significant identifier of reused node, rather than the locally significant one.

Binding Service

Unlike most of the framework-provided services, the binding service is not split into the proxy and workers' parts. Its task is to communicate with instances of publish and subscribe services to make them use their local pipe services for communication channel creation. Any communication channel contains at least one sender and one receiver of messages. In order not to make the processing components buffer the messages, the channels are created "backwards", i.e. starting from the message sink, through message processing components, to message sources. Due to the reused JXTA middleware requirements, each channel is also created starting from the receiving side. Fig. 54 illustrates the generic sequence of calls that are performed by the binding service in order to create a single channel.

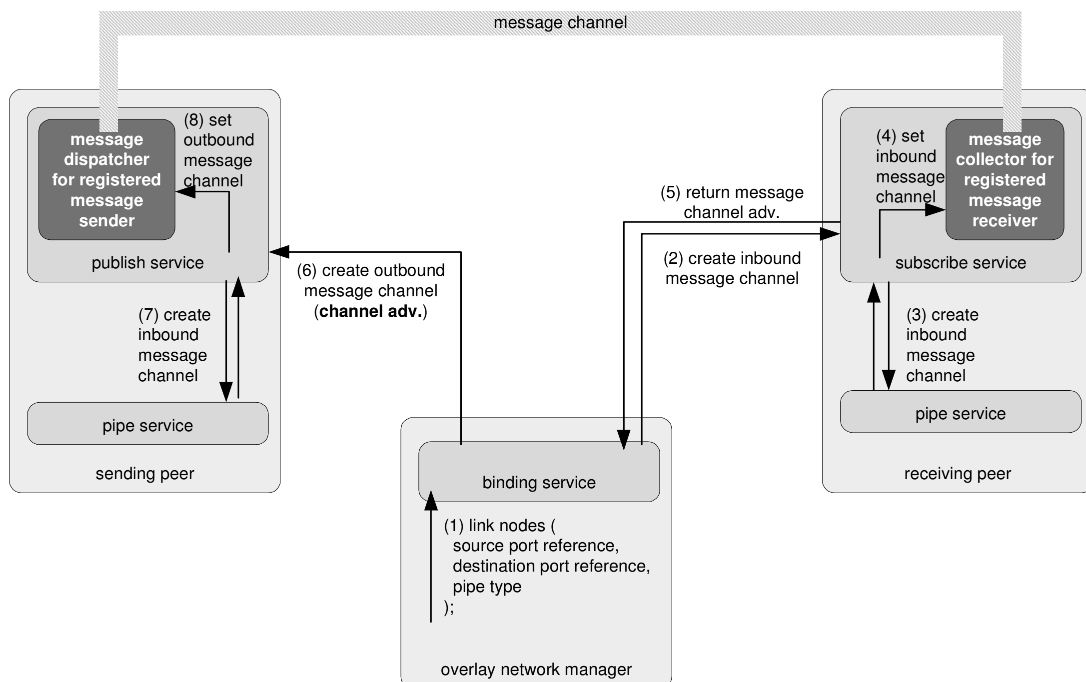


Fig. 54. Creation of a single message channel (simplified).

The prototype implementation of binding service allows also for reusing already created channels. In order to reuse a channel, binding service asks the sending side for advertisements of channels that meet the criteria specified by the user. If at least one channel to be reused exists, the advertisement is passed as a parameter to the "create inbound channel" subscribe service call. The rest of the binding sequence stays unchanged.

Messages from the message sources begin to flow through the overlay network just after the message channels are created. Because all the processing components are initialized before channels creation and the channels are created starting from the message sink, the overlay network starts to provide the virtual producer functionality requested by the message consumer application.

4.4. Key internal services

This section describes most important internal services not directly related to implementation of the key use cases:

- monitoring and metering service,
- redundancy service.

In the framework prototype, the services are used primarily by overlay network managers. The overlay network management service, which plays the most important role in organizing the processing, is implemented by a set of overlay network managers. User applications call the service indirectly, using the requests registered by message sinks. Overlay network managers autonomically decide when and which request should be processed. Such approach was chosen in order to restrict the freedom of overlay network implementations as little as possible.

The manager coordinates the work of most of the framework internal services. In the prototype implementation, each peer group is expected to have a few managers, one of them being elected as active (the election is performed by redundancy service, described later). In case of the active manager getting disconnected, the election happens again between the remaining ones. Because the election may take some time, the redundancy service may improve convergence by electing not only the active but also standby manager, which will become active as soon as it notices the disappearance of the active one.

After creating the overlay the active manager uses the monitoring and metering service to monitor the network both by requesting reports provided by peers that take part in the processing or by instantiating remote probes e.g. for actively testing network characteristics such as round trip time.

Redundancy Service, Service Redundancy Protocol

The task of redundancy service is to provide similar entities with means to elect the active and standby entities. The service covers the details of the underlying redundancy protocol, which keeps one or more backup services informed about the state of the active one and ready to be activated in case the active instance gets disconnected. There are a number of protocols that are designed to accomplish that, e.g. Cisco's Hot Standby Router Protocol (HSRP), Gateway Load Balancing Protocol (GLBP), Virtual Router Redundancy Protocol (VRRP) and Common Address Redundancy Protocol (CARP), existing in computer networks.

The prototype implementation of the service redundancy protocol (SRP) is similar to the multigroup HSRP approach and is suited primarily for peers exposing multiple services that need to be backed up. Multiple instances of the service form a *service group.* In each group, one active service instance is elected. Another instance is a primary backup and is monitoring the active instance's presence and state in order to take over its responsibilities just after the active instance becomes unavailable (e.g. turned off or disconnected).

The service group uses a common communication channel in order to send state synchronization messages (serving also as keepalives), called *hello*s. The active instance is required to send periodic *hello* messages in order to assure the other instances about its presence. The *hello* messages carry the following information:

- protocol version,
- node ID,
- state of the node,

- maximum interval between subsequent *hello*s,
- metric assigned to the instance issuing the *hello*,
- protocol options,
- service-specific data.

The use of the „protocol version" field is obvious. In the future, nodes which use different versions will be required either to negotiate the version to use or – at least – to report the incompatibility.

The "node ID" field identifies the instance of service that issued the hello. Once the state of the service group is established, the nodes to issue such messages would be the active and standby ones. Other nodes are required to listen to the messages and react only if the group needs to reconfigure itself.

Each of the service instances (nodes) goes through several states depending on the service configuration as well as the states of other nodes, especially the active and standby ones. There are five states of the node in a converged group:

- **Active –** the service processes the requests that are coming from external entities and sends out the results of the processing; it also sends periodic hello messages,
- **Standby –** the service processes the requests that are coming from external entities or synchronizes its state with the active instance, but does not send out the results; it also sends periodic hello messages,
- **Listen –** the service does not process any requests coming from outside; its only responsibility is to listen to the hello messages coming from the active and standby nodes,
- **Inactive –** the service is administratively disabled and does not process any messages,
- **ErrDisabled –** the service encountered an error and was therefore disabled; this state is similar to *inactive*, but entered differently.

The state diagram for an SRP node is presented in Fig. 55.

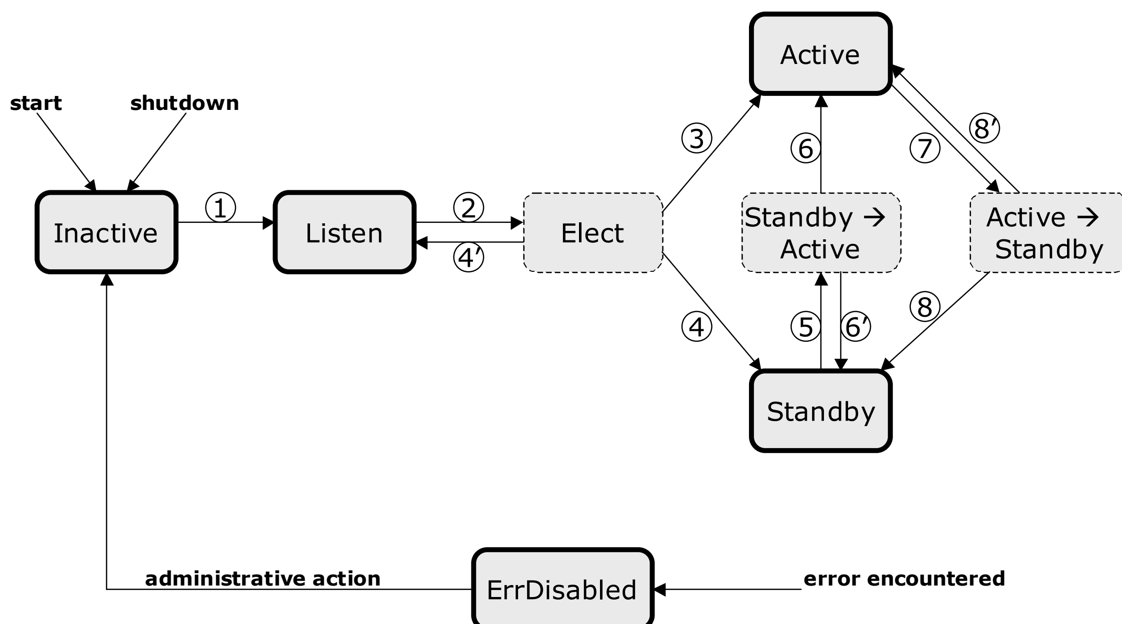


Fig. 55. State diagram for an SRP-enabled node.

The selection of the active and standby instances is based on their current metrics. The metrics can be changed dynamically in order to let the standby group adapt to the changing network conditions. For example, an SRP user may choose the metrics to reflect

some real world value, e.g. the load of the machine the instance in question runs on. Another factor that may influence the metric is e.g. the round trip time measured for messages going to and from a neighbor node. That scenario can be applied to the message processing services. Because the metric may reflect different real life conditions, there is no default implementation of the metric provided by the SRP. The change of the metric assigned to particular service instance may result in reconfiguring the whole standby group by selecting different active and standby instances. In fact, most of the events presented in Fig. 55 are related to metric changes. The events are defined as follows:

1. the service is enabled by the administrator; it enters the Listen state;
2. if for a specified period of time (30 seconds by default) the instance does not hear any *hello* from an active or standby service, the instance enters the temporary Elect state; in the Elect state it sends its own *hellos* in order to start the election of the active and standby nodes;
3. if for a specified amount of time (30 seconds by default) the instance does not receive any *hello* advertising an instance with better metric, it enters the Active state; in the Active state it continues to send out its own *hellos* every 10 seconds by default;
4. if during its Elect state the service notices exactly one source of *hellos* with a better metric, after waiting to the end of the election period it enters the Standby state; in this state it also continues to send out *hellos* every 10 seconds by default;

4’. if two or more sources of *hellos* are heard, the instance falls back to the Listen state;

5. if the standby instance notices that the active instances stopped to send *hellos* or its metric is significantly better than of the currently active one (that can happen either when the active instance’s metric drops or the standby metric raises), it indicates in the *hellos* that it is going to enter the Active state; its state is now Standby → Active; the minimum difference between the metrics is defined as an instance configuration parameter, so that the Active/Standby instances do not swap their roles too frequently;
6. after a configured period the standby service finally enters the Active state; the delay allows for exchanging additional state synchronization information between the instances swapping their roles;

6’. if the metric of the standby instance goes back below the metric of the active instance during the delay, the state is changed back to Standby;

7. if an active instance receives the takeover request from another instance that has a better metric and is going to change its state from Standby to Active it acknowledges the request by entering the Active → Standby state;
8. when the Active → Standby instance receives the first *hello* from a new active instance, it enters the Standby state;

8’. if for a configured period of time the previously active instance does not hear a hello from the new active instance (see 6’), the instance goes back into the Active state.

If in any state the service encounters a critical condition, the state is changed to ErrDisabled. An administrative action to be taken in order to turn the service back into operation depends on the service and the error.

Monitoring and Metering Service

The JXTA platform provides a way for obtaining peer statistics, as well as statistics of monitored modules, which functionality was implemented by the authors of JXTA Metering and Monitoring Project [JXTA-met 2003]. However, the design of that monitoring architecture currently does not fit the framework. The main drawback of the platform approach is its integration with the Peer Info Service that makes the monitored peer return unnecessary data and causes unnecessary processing. Moreover, the JXTA metering subsystem collects the reports in regular intervals, which functionality is unnecessary, especially if generation of a report may involve remote calls.

The monitoring and metering service implemented by the REMP framework prototype is available to any member of the peer group the framework is run in. It provides for gathering runtime information about modules that run on a particular peer. In order to be queried, a module must be registered in the service (typically at the module startup). By default, each of framework services' workers are registered in the service in order to provide information needed by respective proxies. To support reflection, the monitoring and metering service is also registered in itself. A query directed to the module representing remote instance of the service results in a list of registered modules. Fig. 56 shows the process of registering a module to be monitored and obtaining the module's report.

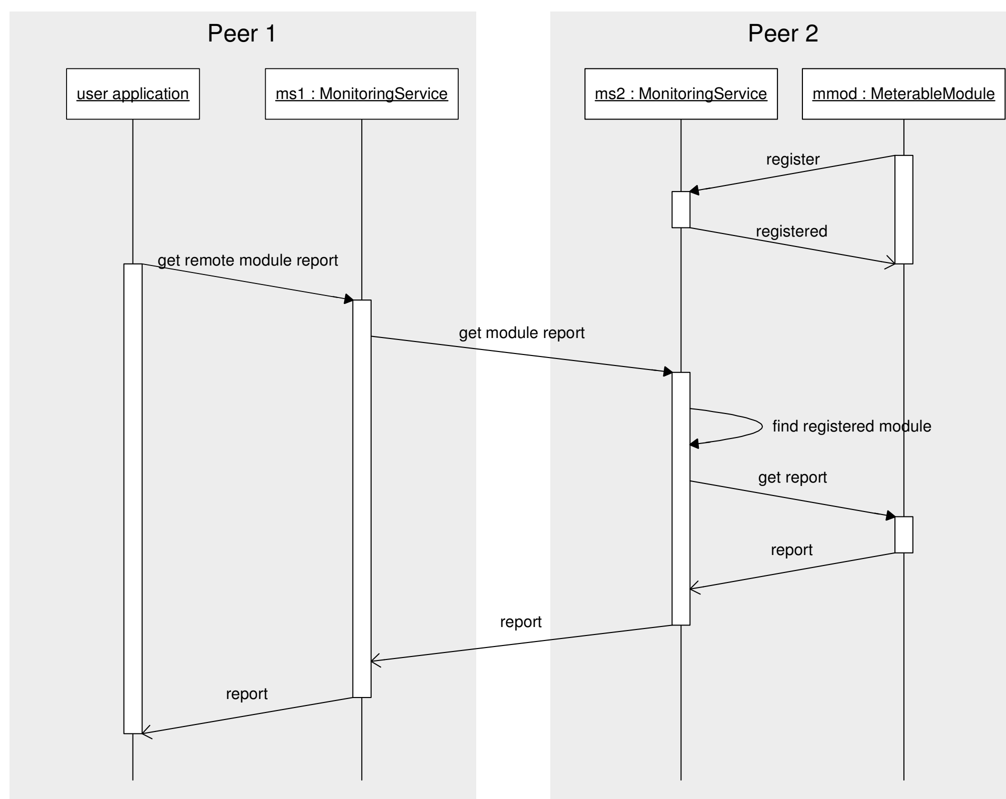


Fig. 56. Generic sequence diagram for obtaining remote module reports.

The query issued by remote peer must include a reference (identifier) of the target module, which defaults to the module specification ID. It may also contain additional query that is passed to the module (useful e.g. if the target module can generate various types of reports). By default, the Monitoring Service provides also means for measuring the round trip time between any peers running its instances by registering a dedicated module that in the "get report" method calls another peer's Monitoring Service. Such functionality can be

of help for the Deployment Service, when deciding a deployment option to be executed or to an overlay network manager, so that it may react to topology changes.

4.5. Summary

The chapter described the key REMP framework mechanisms mainly from the user perspective. The prototype provides core functionality needed to create networks specified by framework instance users. Although the functionality may be provided by as little as six peers, the implementation includes means for operating in much larger networks. The choice of implementation technologies influenced the implementation both in important details (such as addressing) and by defining the scope of services availability (a single peer group). Nonetheless, the implementation was not severely affected by the constraints as none of the assumptions made at the framework model design phase has been violated.

Prototypes of the services implement the functionality identified in Chapter 3 to various extents. Functionality provided by most of them is quite extensive, while some are restricted to core functions. Key lacking features include security mechanisms (containers) and partial network recovery (overlay network managers). Because the goal was to provide prototype for a framework, not for its instance, the lack of sophisticated deployment strategies is not considered a crucial disadvantage, provided that discovery and monitoring services expose interfaces for supporting them.

One of the key assumptions was to make the framework prototype extensible. Most of the services expose interfaces for custom extensions, although some of them require at least basic knowledge of framework internals. The proxy-worker pattern used by most of the framework services provides much freedom in implementing the functionality of particular services.

The main advantage of the prototype implementation is its simple but powerful user (developer) interface. The internals (protocols and services) are hidden by just six simple and well-defined application programmer interface objects. Moreover, it seems that the interface should not be much more complicated even in more featured framework implementations. Nonetheless, the overlay network specifications are very complex documents, so software support for creating them is certainly desirable.

Chapter 5 presents a use case that served to evaluate the framework prototype in context of a real-life application.

5. Evaluation of framework prototype

The prototype of the framework was intended to serve more as a proof of concept than as a fully functional environment. Therefore, the implementation is focused more on extensibility than efficiency. In order to evaluate the prototype, a few case studies were designed and implemented. Among the implemented ones, "marathon reporting service", described in section 5.1 is the one that uses most of framework prototype features. Section 5.2 describe other use cases that might be implemented either with the prototype (stock monitoring) or with the fully functional framework implementation (accident rescue system). Section 5.3 summarizes the chapter.

5.1. Marathon reporting service case study

City marathons are sport events that could be challenging not only for the contenders, but also for the observers, especially for the remote ones. First of all, taking into consideration the distance of the race, the numbers of contenders are really impressive. That makes the online tracking of a subset of participants challenging, especially when the participants' paces vary greatly. In order to feed tracking systems with data, the athletes are required to wear transponders that allow sensors located at specified points of the course to report mid-times of particular contenders (the same mechanism could be used to ensure the more city-oriented athletes do not take shortcuts).

Sensors send out raw data (participant identifier and current time). The messages to be read by human users should carry more information, including the participant's of interest name, surname, club, nationality, category, etc. Therefore, a contenders database must be consulted before sending the message to the internet. The database design is not very interesting, because it does not undergo any modifications once a race starts. Therefore, if one instance is not efficient enough, simple replication will satisfy its users.

What is interesting is how to process the large numbers of messages so that the clients are not flooded with unnecessary ones. In an extreme case, a person interested in watching a family member participating in a race of 20.000 contenders watched by 20 sensors could get 400.000 messages, including 20 relevant ones (0,005%). Given that the reliability of communication over WAN is significantly lower than 100%, there is a chance that an interesting message is dropped (either due to an overload or other network factors). A typical human interpretation of such "no message" message might suggest that the observed contender is hurt, or that has quitted the race for other reason. Of course, no clever design of an application cannot assure the WAN links to deliver every message at first attempt, but at least should not cause the congestion by itself. Rather, it should filter out the irrelevant messages as soon as possible.

Typical Internet "live score" services based on polling respective servers refresh the data every 45-90 seconds (e.g. f1-live.com, livescore.com, etc.). It seems that efficiency is not the reason to do that. Although the messages sent to the client are aggregated into a single bundle, they must undergo transport-layer segmentation, so the aggregation does not allow for saving much on network-related overheads, especially in case of elaborate messages to be read by humans. It seems that the reason for aggregation is that a bundle is typically formed as a web page that contains not only the important data, but additional content, such as advertisements, as well.

Implementation of a single-contender tracking application is quite straightforward and does not provide much room for showing the framework functionality. Therefore, the case study focuses on simulating a system for delivering relevant messages regarding particular marathon race to users of three kinds:

- journalists, who are interested mainly in being notified if somebody from their country was recorded in any "top 10" classification,
- professional runners, who are interested in all "top 10" classifications as well as average paces of the runners during the race,
- club managers, who are interested in knowing the results of the club representation.

As a real life example to be simulated, Milano City Marathon 2008 was selected. Section 5.1.1 introduces the types of consumer applications implemented for the case study as well as the topology of overlay network that was created to serve the consumers. Section 5.1.2 assesses the REMP prototype in terms of usability from the marathon reporting service developer's point of view and shows the most important steps of implementation. The following section, 5.1.3, describes the network topology that was used to run the evaluation-related experiments. Section 5.1.4 presents the experiments that were conducted. Finally, section 5.1.5 discusses the results of the evaluation.

5.1.1. Marathon reporting service model

In order to design a model for an overlay network that can serve many users, this section starts with analysis of requirements and logic of the selected classes of reporting service clients. Then, an overlay network model that satisfies the requirements is presented.

Journalist's application. The basic requirement of an imagined journalist is to receive any "important" result of a marathon race participant. For the purpose of the simulation the term "important" is defined as:

- "in top 10 results", or
- "in top 10 results in either male or female classifications", or
- "in top 10 results in participants' category".

In the analyzed Milano City Marathon, 23 categories of participants were defined. Assuming that all categories contain at least 10 participants, the number of messages to be received from the first stage filter, that checks the raw results against the criteria of "importance", should vary at maximum from 260 to 290 (if no more than 10% raw messages arrive out of order) per sensor. Comparing to the original number of 5000 messages per sensor, the filter reduces the volume of messages by about 95%.

The second stage of filtering refers to the nationality of the runners. The percentage of message volume decrease at that stage may be even higher – for example, no Polish runner was present in any of the "top 10" classifications at Milano, so the reduction can reach 100%, meaning that the journalist should look for some other topic for his article. On the other hand, from the top 10 runners, four came from Kenya. Moreover, the "top 10" classifications in categories MM40 (male runners at age from 35 to 40) and MM45 (males, from 40 to 45) were dominated by Italians – no runner from any other nation was recorded there. In such case, the reduction is significantly lower. Taking into account that 260-290 messages per sensor means less than 12000 messages in total during six hours, the real saving on network bandwidth and processing time is low, when compared to the first stage.

Fig. 57 illustrates the logic of journalist's application, which is built from three components representing respective message processing stages. Because from the user's point of view it would be convenient if the end user application was run on a limited device,

e.g. mobile phone, restricted both in terms of available memory and computing power and connected to the network by a low-capacity link, it would be desirable to move as much message processing as possible to the network. Therefore, Fig. 57 suggests also that all the components should be delegated to the REMP instance as message processing nodes of the overlay network to be deployed.

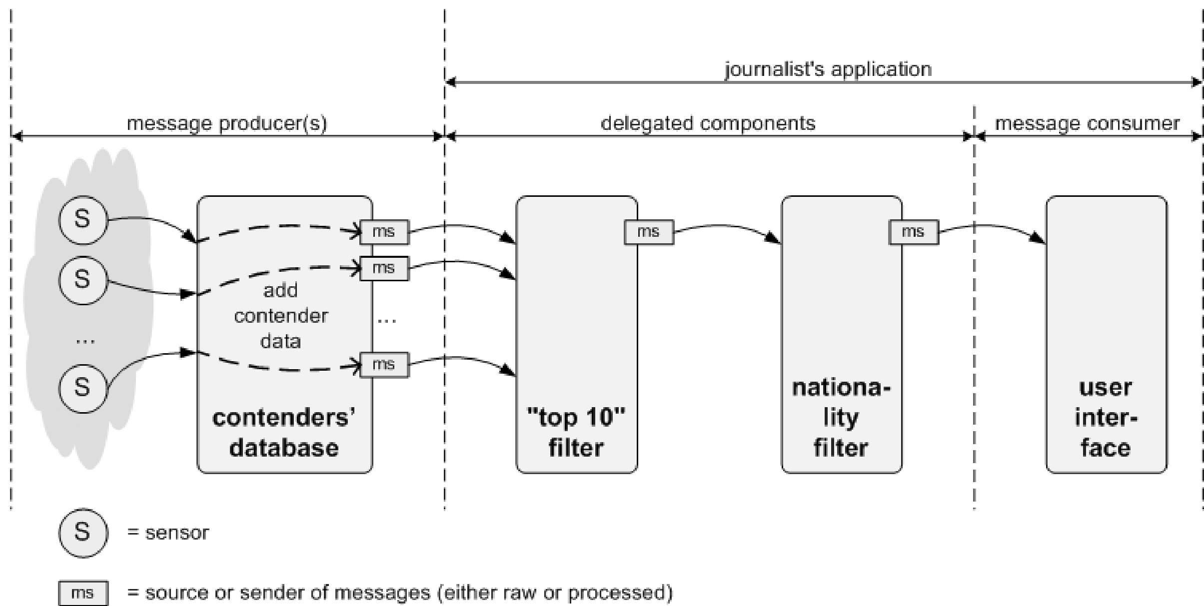


Fig. 57. Components that build up the journalist's application.

Professional runner application. Sportsmen form the second class of the reporting service users. The imagined professional runner is interested is to know who was best in particular category and – more importantly – in the average paces of first 10,100 and 1000 runners. Therefore, the functionality to be delegated is not limited to message filtering, but includes message processing as well.

Fig. 58 depicts the logic of the professional runner application. The main difference between the journalist's and runner's applications is that in the latter case the two processing components operate in parallel, not sequentially.

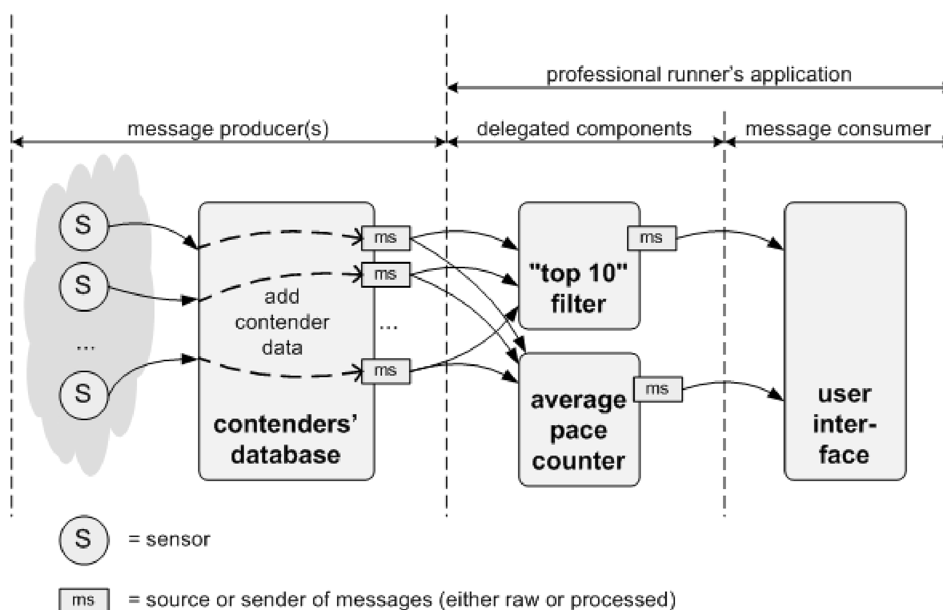


Fig. 58. Components that build up the professional runner's application.

Club manager application. The club manager application is the simplest one – it implements only one filter that can be delegated. Of course, real club managers need to know more than the results of their representatives (for example, the averages counted by the professional runner's client would also be of interest to the managers), but in order to demonstrate the application creation process and perform baseline measurements, functionality of that application was deliberately reduced to the absolute minimum (Fig. 59).

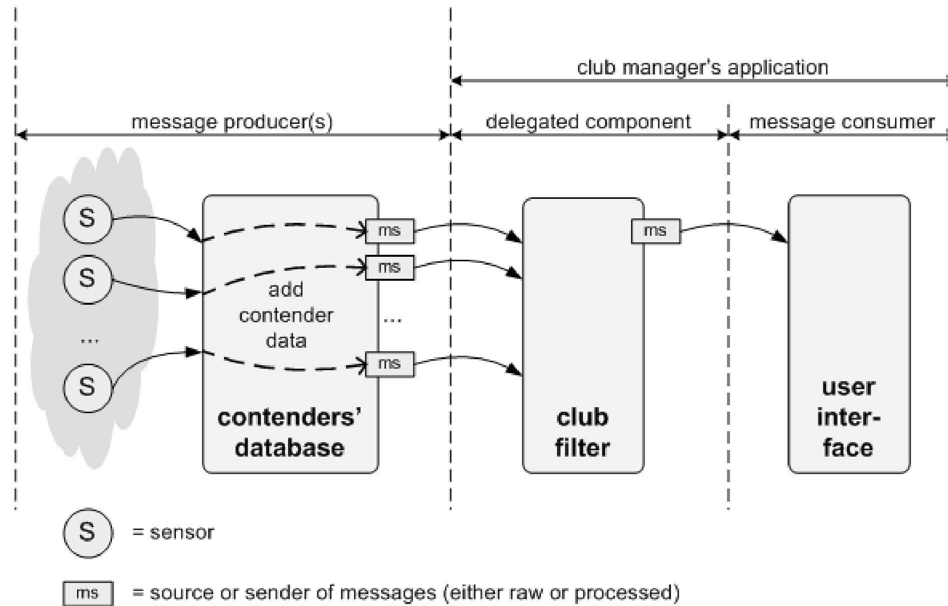


Fig. 59. Components that build up the club manager's application.

Contender fan application. The club manager application can be easily extended to a single-contender watcher application, to be used e.g. by the contender family members and fans. The messages published by club filter could be further filtered by a "person filter", as depicted in Fig. 60. Similarly to the journalist's application scenario, the second filter will be fed with just a few messages (from each sensor) – most of the work will be performed by the re-used club filter.

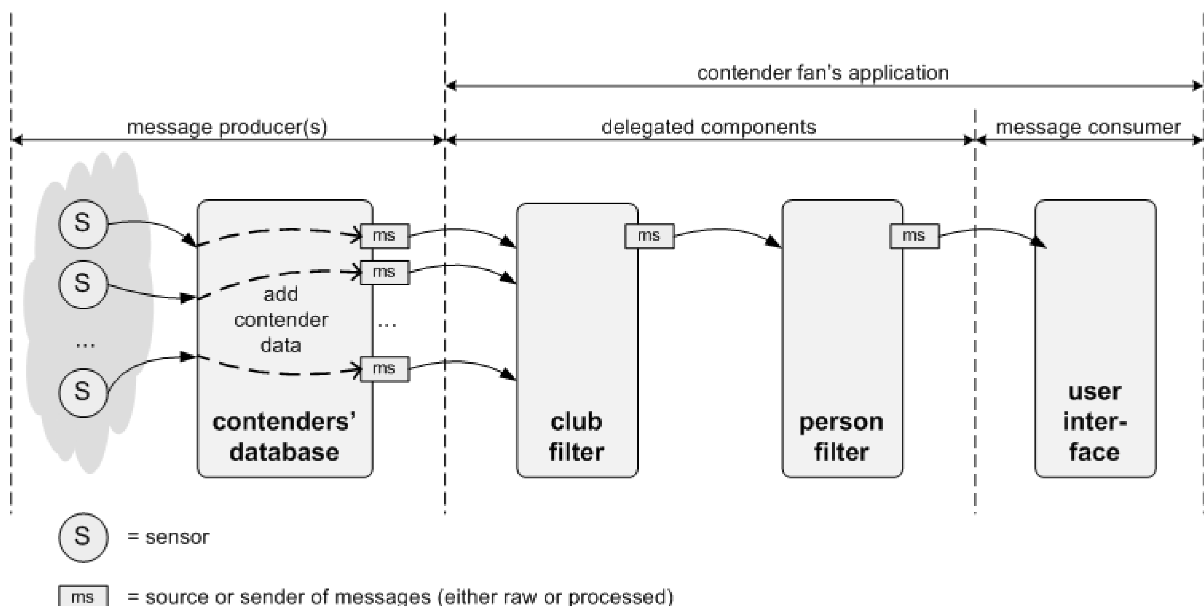


Fig. 60. Components that build up the contender fan's application.

Of course, there are a lot more of potential client types. These were selected to show that appropriate design of an application along with the features provided by the framework

prototype allow to serve much more clients than using the naive flooding-based approach. As a result of combining the requirements of the three application types, the overlay network depicted in Fig. 61 was designed.

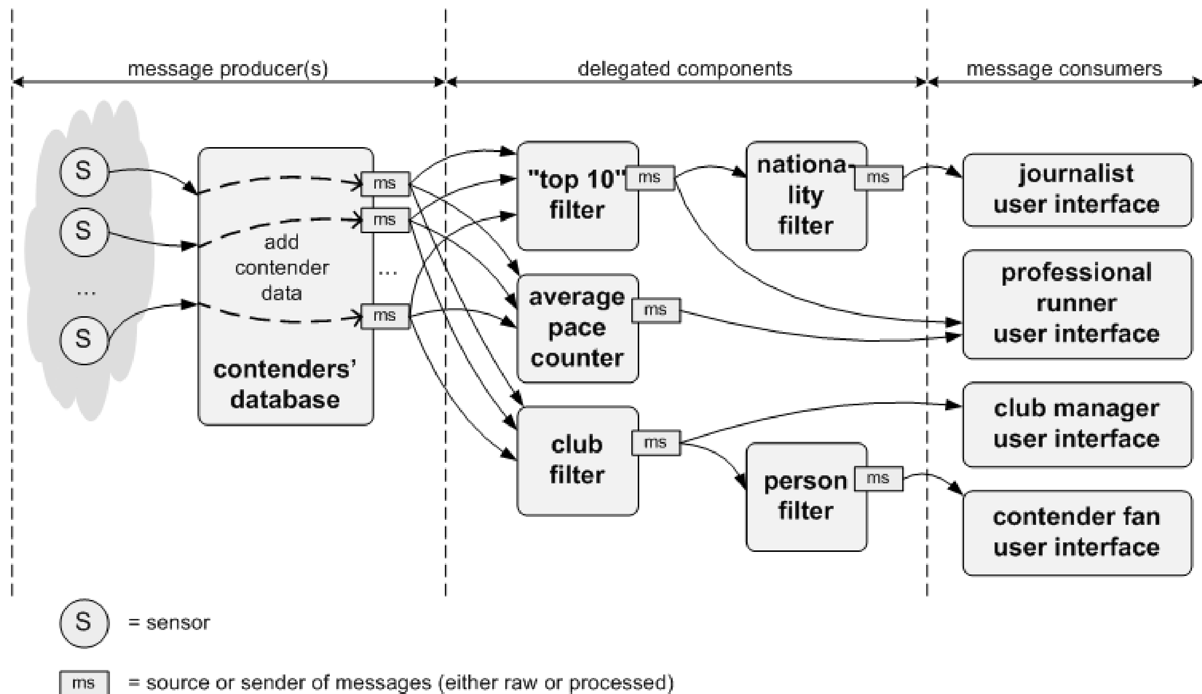


Fig. 61. Marathon reporting service overlay network.

The marathon reporting service overlay network is expected to provide truly live service, without periodic polling and message aggregation. Moreover, the network can be extended at runtime to provide new filters/processors of messages. Note that, from REMP's point of view, the reporting service is provided by many cooperating overlay networks (a separate network is created by each message consumer), so the term "marathon reporting service overlay network" used in this section should not be read literally.

5.1.2. Implementation of the service based on the REMP prototype

The goal of this section is to assess the REMP prototype ease of use from the service developer's point of view. Such assessment is performed by the developers with regard to any of the frameworks they use on a daily basis, and frequently is the main factor that guides their choices. Because only a prototype of the framework was implemented, the goal of this section is not simply to say if REMP is easy or not easy to use, but as well, what is missing and should be implemented in the future to support the users better.

Implementation of message producers

According to the definition introduced in section 1.1, a message producer is "an actor (user application) that produces messages to be processed by the message processing system". As described in section 3.2.1, a producer registers one or more offers for contracts for message publishing using its local instance of the Publish Service provided by the framework. Section 4.3.1 briefly describes the Publish Service interface as implemented in the REMP prototype.

Technically, the message producer application constructs one or more instances of objects that implement the MessageSource interface. To simulate the raw message sources present in the marathon reporting service overlay network, an application that registered a

configurable number of message sources was developed. The application configuration parameters include the number of sensors to be simulated as well as the results of selected contenders (see Listing 5). The results of contenders that are not present in the configuration file are interpolated linearly based on the pre-set ones.

```
# file: sensor.properties.40k10s - NYC Marathon 2008
# 40000 participants, 10 sensors

# delay between publishing advertisements of sources and first message
startDelaySeconds = 180

# sensors are numbered from 0 to the "sensors" property value -1
sensors = 10

# syntax for sensorN property:
# sensorN = <distanceInMeters> {<position:timeInSeconds>}*
#
# the last position:time entry denotes the last contender
# that reached the sensor

sensor0  =     0  1:0     1000:80    10000:240   40000:600
sensor1  =  5000  1:960   1000:1320  10000:1740  40000:3000
sensor2  = 10000  1:1860  1000:2400  10000:3000  40000:6000
sensor3  = 15000  1:2820  1000:3600  10000:4800  40000:9000
sensor4  = 20000  1:3720  1000:4800  10000:6300  40000:12000
sensor5  = 25000  1:4680  1000:6120  10000:8100  40000:15000
sensor6  = 30000  1:5520  1000:7500  10000:9840  40000:18000
sensor7  = 35000  1:6420  1000:8940  10000:11460 40000:21000
sensor8  = 40000  1:7320  1000:10260 10000:13140 40000:24000
sensor9  = 42195  1:7740  1000:10800 10000:13860 40000:25500
```

Listing 5. A sample configuration file for the message producer application.

After startup, the application registers one message source object for each of the sensors to be simulated. Because the sources are advertised separately, consumer applications refer to them independently. In the extreme cases, one producer application can represent either one sensor or all sensors without introducing any implications for the consumers.

Message sources are required to fill up the MessageSourceCapabilities XML document, depicted in Fig. 39 (in section 4.3.1), that describes the registered offers. The producer application uses XMLBeans library for that purpose. Because the document contains only five fields, three of them having sub-fields, it could as well be filled up with an editor and read from a configuration file. However, because the producer application has to register many sources and therefore create a series of such documents, API-based approach seems to be more reasonable. The most important fields of a MessageSourceCapabilities are:

- semanticMessageDescription, which a description of message semantics,
- messageSyntaxDescription, which carries a description of message syntax.

The descriptions of message semantics and syntax can be multiplied, and expressed in various languages. For the marathon reporting service producer application, OWL for message semantics and XML Schema for message syntax were chosen.

The semantic description of messages is very simple. A single individual of a class MessageSource represent a single individual of class Sensor. Sensors produce Results, which are included in Messages (Fig. 62).

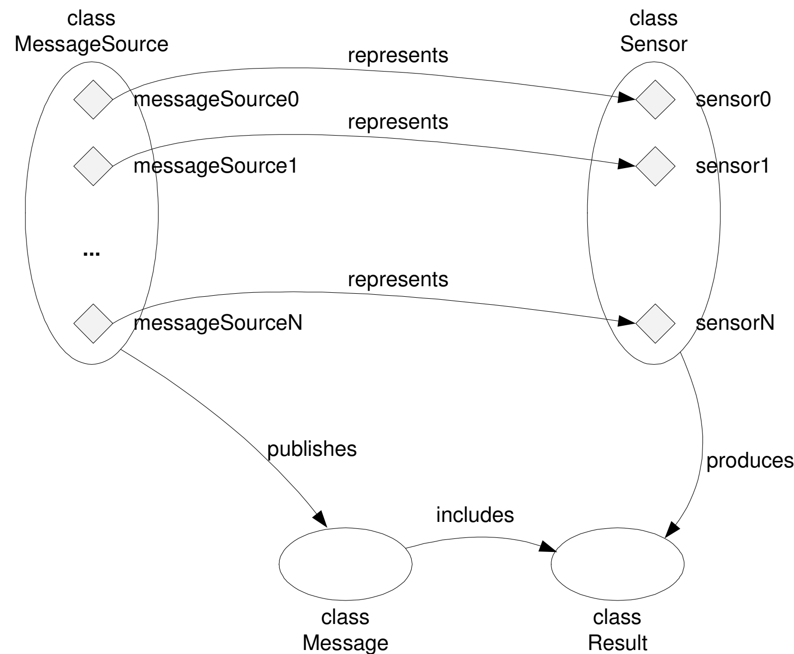


Fig. 62. The ontologies published by message sources.

Even that simple ontologies are hardly editable in a text editor. Therefore, a generic document was created in an OWL editor. The producer application, on the configuration file basis, just filled up the gaps, i.e. properties:

- “locatedAt” for a sensor,
- “number” for a sensor,
- “number” for a message source”.

Such representation was enough for consumers to ask for results from specific sensor, which is just enough for the clients of the designed service.

The description of message syntax was expressed by an XML Schema document, containing a field of XML Schema type presented in Listing. The elements of the message are based on the reports published for the Milano City Marathon.

```
<xs:complexType name="ContenderResult">
  <xs:sequence>
    <xs:element name="startNumber" type="xs:int"/>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="surname" type="xs:string"/>
    <xs:element name="club" type="xs:string"/>
    <xs:element name="nationality" type="xs:string"/>
    <xs:element name="MW" type="marathon:ManWoman"/>
```

Listing 6. The elements of a single contender result message (continued on next page).

```
    <xs:element name="category">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="TM"/>
          <xs:enumeration value="TF"/>
          ...
          <xs:enumeration value="MM80"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="distanceInMeters">
      <xs:simpleType>
        <xs:restriction base="xs:int">
          <xs:minInclusive value="0"/>
          <xs:maxInclusive value="42200"/> <!-- should be 42195 -->
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="position" type="xs:int"/>
    <xs:element name="timeInMillis" type="xs:long"/>
    <xs:element name="realTimeInMillis" type="xs:long"/>
  </xs:sequence>
</xs:complexType>
```

Listing 6. The elements of a single contender result message (continued).

The semantic and syntactic descriptions of messages form the main parts of the `MessageSourceCapabilities`. There are three other parts in that document: `humanReadableDescription`, which serves as a place for short documentation-related notes, `portName`, which is filled with the string “sensor<M>” (M being the value of sensor’s `locatedAt` property), and `supportedPublishingPolicy`, which refers to the offered message publishing model (push, pull or periodic). Regarding the quality of service, the sources do not leave any choice for their potential counterparts, and offer only the “push” scheme of delivery with regard to single messages. For a real-world application that seems to be a poor choice, especially when considering that JXTA and lower layers add an overhead of over 1kB to each message which size is just over 750B on average. However, the goal of the simulation was to evaluate the framework prototype, not the application, and therefore the experiments are based on a simpler approach.

At start, the producer application looks for a contenders database file. If no such file is present, it is generated on a random basis. After obtaining contenders’ database, the application registers a series of message sources and schedules execution of threads that simulate particular sensors. Because J2SE environment does not provide any real-time guarantees, the messages published by the simulated sources were logged just after the message publishing in order to check if delays between messages were significantly different than scheduled, which could result from an overload. In such case, the responsibility of publishing messages should be distributed among larger number of sources.

Implementation of message processing components

Message processing components represent well-defined parts of message consumers’ functionalities to be delegated to the overlay network. Therefore, the

development of consumer applications could be started with implementation of the processing components. Based on the marathon reporting service overlay network topology (Fig. 61), five processing components were identified:

- “top 10” filter, that maintains current “top 10” classifications, based on raw messages,
- average pace counter, that counts average paces of first 10,100 and 1000 runners at each of the checkpoints (sensor locations),
- club filter, that forwards the results of contenders that are members of a particular club,
- nationality filter, that based on the messages from “top 10” filter, selects the results of contenders of particular nationality,
- person filter, that based on the messages forwarded by the club filter, forwards only the results achieved by a single person.

This section presents the implementation of the “top 10” filter, that was selected, because it operates on many raw message streams, and because its results are used both by other message processing components and user applications.

The interface to be implemented by a message processing component was presented on Listing 2, in section 4.3.2. It consists of two methods only, one for starting up the component, the other for receiving messages from declaratively specified ports. At startup, the component is given a reference to its wrapper and an implementation advertisement, that may contain the component configuration. In the “top 10” filter’s case, the configuration argument is not used. The filter was implemented as an ordinary Java class. Listing 7 contains excerpts of the filter’s code which refer to the container-component communication.

```
public class Top10Filter implements remp.service.deployment.Component {
  static final String best10ReportsOutputPortName = "best10ReportsOut";
  ComponentWrapper componentWrapper;
  private HashMap<String,Classifications> classificationsByPortName;

  public void startProcessing(ComponentWrapper wrapper)
  throws remp.service.deployment.ComponentStartupException {
    componentWrapper = wrapper;
    ...
    String[] inputPortNames = componentWrapper.getInputPortNames();
    for( String ipn: inputPortNames ) {
      classificationsByPortName.put( ipn, new Classifications(ipn) );
    }
  }

  public synchronized void newMessage(String inputPortName,Object message){
    ...
    try {
      ContenderResult result =
        ContenderResult.Factory.parse(message.toString());
      Classifications cl = classificationsByPortName.get(inputPortName);
      if( cl!=null ) cl.processContenderResult(result);
      ...
    }  catch (XmlException ex) {...}
  }
```

Listing 7. Excerpts of the “top 10” filter code.

Writing message processing components' code is probably the easiest part of the application (and overlay network) development. In this sense, the REMP prototype fully satisfies the "user friendliness" requirement. Note only, that the code uses some "background knowledge", i.e. it exploits the knowledge of message type, which is passed to the component as an ordinary `java.lang.Object`. Such approach might be controversial, but it stems for the extensibility requirement, which in this case means avoiding anything that could restrict the set of message representations. If the Component interface contained several overloaded methods e.g. for `Serializable`, `XmlObject` or other interfaces, introducing a new representation would require the developer to change the framework's source code.

The source code is just one part of the processing component specification (Fig. 47 in section 4.3.2). Note, that the code uses some literals that denote inbound and outbound ports. The literals must match what the developer places in the `inputPortSpecification` and `outputPortSpecification` elements (Fig. 48 in section 4.3.2). Writing the code that fills up the descriptors is a lengthy process, compared to what a developer would expect, when looking on the figures. Again, if the number of sensors present in the simulation scenario was fixed, an XML-enabled editor could be used for the purpose. However, given that that number varies, the only way is to use an API-based approach. The code that fills up just a single inbound port descriptor is listed in Listing 8.

```
static InputPortSpecification createRawMessagesIPS( Params params ) {
  SourceSelectOperation selop =
    SourceSelectOperation.Factory.newInstance();
  selop.setComponentNumber(100+params.sensorNumber);
  selop.setLinkCriteria(createSourceLinkCriteria());
  selop.setMessageSourceSelectionLanguage("SPARQL");
  selop.setSelectExpression(
    "PREFIX marathon: <http://www.cs.agh.edu.pl/~slawek/remp/marathon_cs>"+
    "SELECT ?adv WHERE "+
    "(?x remp:hasAdvId ?adv)"+
    "(?x rdf:type remp:MessageSource)"+
    "(?x marathon:represents ?y)"+
    "(?y rdf:type marathon:Sensor)"+
    "(?y marathon:locatedAt "+params.distanceInMeters+")"
  );
  MessageDescription syntDesc = MessageDescription.Factory.newInstance();
  syntDesc.setMessageDescriptionLanguage("XMLSchema");
  syntDesc.setMessageDescription(readFile(RESUL_SCHEMA_FILE));

  InputPortSpecification ips =
    InputPortSpecification.Factory.newInstance();
  ips.setPortName( "raw messages from sensor "+params.distanceInMeters );
  ips.setLinkOperation( selop );
  ips.setMessageSyntaxDescription( syntDesc );

  return ips;
}
```

Listing 8. The code that fills up one inbound port specification (link criteria omitted).

The InputPortSpecification contains a SourceSelectOperation, which defines a concept-based query expressed in SPARQL, referring to the raw message source of interest. Noteworthy, the query should return the identifier of an advertisement of message source capabilities. The identifier is used by the overlay network manager to obtain a full advertisement from its instance of JXTA Discovery Service. However, taking such approach seems to break the REMP instance's opacity, because the advertisements are published by the Publish Service, not directly by user applications, and therefore – theoretically – the developer could use the framework without knowing anything about advertisements.

An alternative approach would be to hide the "SELECT ?adv WHERE (?x remp:hasAdvId ?adv) (?x rdf:type remp:MessageSource)" part from the users. That part should be present in all queries served by the Matching Service worker provided with the prototype. It was left visible mainly because writing SPARQL queries without the SELECT and WHERE keywords could confuse the developers. Luckily, the syntax of the concept-based queries depends on what is served by the Matching Service workers, so the option of introducing a new worker, which adds the constant prefix to the query before its execution, remains.

In REMP, there are two ways of supporting reuse of the computing results: by advertising an outbound port as a message source, or by handing the processing component UUID to the reusing one in order to connect using ExternalLinkOperation, as described in section 4.3.2. The "top 10" filter component uses the first approach, by placing MessageSourceCapabilities instead of its superclass OutputPortSpecification in the outboundPortSpecification (Listing 9).

```
static OutputPortSpecification createDelegateBest10OutPortCapabs() {
  MessageSourceCapabilities best10sc =
    MessageSourceCapabilities.Factory.newInstance();
  best10sc.setPortName(best10ReportsFilterOutputPortName);
  best10sc.setHumanReadableDescription(
    "Messages for various \"best 10\" classifications.");
  MessageDescription syntaxDesc =
    best10sc.addNewMessageSyntaxDescription();
  syntaxDesc.setMessageDescriptionLanguage("XMLSchema");
    syntaxDesc.setMessageDescription(readFile(TOP10_MESSAGE_SCHEMA_FILE));
  MessageDescription semanticDesc =
    best10sc.addNewSemanticMessageDescription();
  semanticDesc.setMessageDescriptionLanguage("OWL");
  semanticDesc.setMessageDescription(readFile(TOP10_MESSAGE_OWL_FILE));
  return best10sc;
}
```

Listing 9. Creation of a message source specification.

Overall, filling up the declarative parts is much harder than writing the message processing component's code. Even for quite simple components, such as used in the use case, the declarative part takes a lot of development time. Clearly, the framework prototype lacks of a tool that would generate the code to be customized, or at least fill up the documents stored in files.

Implementation of message consumers

The message processing components having implemented, the developer needs to pack their code and port descriptors in a single OverlayNetworkSpecification (Fig. 43), which consists mainly of the sequence of processing component specifications and the specifications of user application input ports. In case of an application that delegates its functionality to the overlay network, the specifications typically contain the simple InternalLinkOperation elements, which carry only the overlay-local address (i.e. the number) of the component to be linked to, its port name and hints for communication channel creation. The overlay network specification is wrapped into message sink object, that represents the message consumer to the REMP instance.

To illustrate this part of application development, the professional runner application was chosen. The application uses two delegated components, namely “top 10” filter and average pace counter. Both components register their output ports as message sources rather than mere publishers, therefore making it easier to reuse the computation results. The end-user application could be configured either to deploy the network, or reuse the existing one. Again, the application could be made intelligent enough to check if the needed sources are present, and if not, deploy them. But – again – the goal of the use case was to evaluate the framework, not the application.

In the network deployment mode, the application creates the processing component descriptors by loading their source code, creating port descriptors, etc. and registers its message sink object, which results in publishing a MessageSinkNeedsAdvertisement by its local instance of the Subscribe Service (see Fig. 42 in section 4.3.2). If only the required elements of a REMP instance (see Fig. 36 in section 4.2) are present, the network is created in accordance with the query. In the reuse mode, the query is much simpler – it contains only the specifications of message sink’s inbound ports and human-readable description (Listing 10).

```
static InternalLinkOperation createTop10InternalLinkOperation() {
  InternalLinkOperation ilop = InternalLinkOperation.Factory.newInstance();

  ilop.setComponentNumber(filterComponentNumber);
  ilop.setLinkCriteria(createBest10LinkCriteria());
  ilop.setOutputPortName(best10ReportsFilterOutputPortName);

  return ilop;
}
```

Listing 10. Specification of the arguments needed to create an internal link (link criteria omitted).

Comparing to the delegated components’ case, filling up the declarative parts of overlay network specification is really easy. Nonetheless, it could be made easier by the aforementioned tool for filling up the required specifications.

Implemented applications

Using the approach described earlier in this section, four consumer applications were implemented: journalist application, professional runner application, club manager application and contender fan application, being actually a variation of the club manager application. In order to facilitate the testing scenarios described in section 5.1.4, the application user could run the applications in one of three modes:

- DIRECT, in which all the message processing was carried by the application, i.e. the application was linked directly to the raw message sources,
- DEPLOY_NETWORK, in which all of the specified processing components were delegated outside the application,
- REUSE_NETWORK, in which the application searched for the (already deployed) components to be reused (in case of journalist's application only the "top 10" filter was reused).

After startup, the consumers read the list of sensors they are interested in, create queries regarding the raw message sources (and processing components) and wait until all communication channels are created. To reuse the underlying platform capabilities, the parameters of the query were adjusted so that so called propagate pipes (which implement a sort of application-layer multicast) were used. Messages published through the pipes carry significant amount of JXTA-specific information, but in return they should require less bandwidth than multiple unicast ones. In case of congestion, the messages may be simply dropped.

5.1.3. Evaluation environment and overlay network startup

The REMP prototype was tested on a dedicated network consisting of 20 PCs split into three separate LANs connected by three routers and two WAN links. Fig. 63 illustrates the hardware configuration. One of the networks was selected as the "producer network", two others as the "consumer networks". The consumer networks were connected to the producer's via 4Mbps and 640kbps links, respectively. For some experiments, the bandwidth of 640kbps was reduced to 64kbps to better illustrate the gains from delegation of the processing.

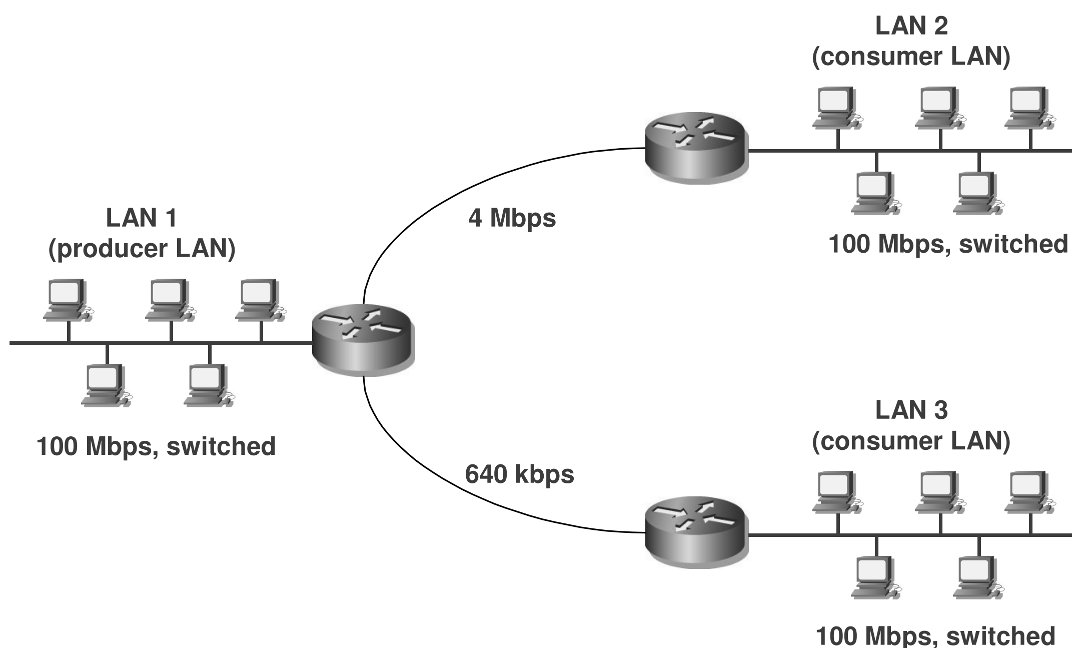


Fig. 63. Evaluation environment network topology.

The PCs present in the networks were uniform both in context of hardware and software configuration. In order to run REMP prototype, each of the hosts was required to have a few software packages installed. Table 3 summarizes the hardware and software configurations of the hosts.

Table 3. Hardware and software configurations of the hosts present in evaluation environment.

Element	Description
hardware	
processor	Intel® Celeron® 2.8GHz, 256 kB L2 cache
Mainboard, chipset	ASUS P5-P800, Intel® 865PE
RAM	1GB DDR-400
HDD	Samsung SP0812N, 80GB, ATA-133, 7200 rpm, 8MB cache
Network interface	Marvell Technology Group Ltd. 88E8001 Gigabit Ethernet
software	
Operating system	Fedora Core 6 Linux, kernel 2.6.18
Java VM	Sun JDK 1.6.0_01-b06
JXTA	2.4.1
XML Beans	2.4.0
IODT Minerva	1.1.2
REMP	0.6

Installation of the required software components is straightforward. For each of them, an archive needs to be unpacked and a single environment variable, e.g. REMP_HOME, XMLBEANS_HOME etc., needs to be set. Based on the values of the variables, REMP components startup scripts are able to construct the class paths needed by the respective applications.

Note that each of the REMP components needs at least one JXTA endpoint in order to communicate with the rendezvous and other components of the P2P networks. The default settings allow to run any combination of REMP infrastructure elements at the same host, but in the real world, the endpoint configuration may need to be tuned. Listing 11 shows an excerpt of a default configuration file for the marathon reporting service producer application. Each of the prototype software elements is configured in a similar way.

```
# PlatformConfig-related properties
remp.reconfigure=false
remp.peerNamePrefix=marathon_sender
remp.peerNameSuffix=hostname
remp.peerDescription=REMP 0.6 Marathon Case Study (Sender)

remp.userName= remp
remp.password= remp123

# endpoints
remp.httpPort=19000
remp.tcpPort=19001
remp.rendezvousURI=tcp://149.156.97.89:17001/
# multicast is rarely supported in WANs
# remp.multicastAddress=239.255.255.77
# remp.multicastPort=7700
# end of PlatformConfig properties

remp.autoRendezvous=false
remp.rendezvousConnectionTimeoutSeconds=60
```

Listing 11. A part of default configuration file for REMP elements.

The producer network hosted all needed REMP infrastructure elements, namely:

- JXTA rendezvous,
- overlay network manager,
- matching service worker,
- component generation service worker,
- code repository service worker,
- deployment service worker (i.e. container for the deployed modules).

The message producer application was run on a separate machine in the producer network as well. Because after the overlay network setup phase most of REMP-internal elements of the infrastructure generate little network traffic, placing them in the same LAN with the producer did not introduce any significant load. The only exception are the containers (deployment service workers), which play an important role in the overlay networks' operation. Note however, that if the deployed message processing nodes are expected to decrease the volume of message streams significantly, having a deployment service worker in the same LAN with message producers is desirable.

The consumer LANs hosted respective consumer applications as well as deployment service workers (one per LAN). Table 4 summarizes the configuration of the REMP instance that was used for the evaluation. The default implementation of the deployment service places the processing components as close as possible to the message sources, which is right in most cases, when the processing reduces the amount of network traffic. The distance between containers (deployment service workers) and sources is measured by using the metering service to issue a series of round-trip-time tests. In order not to select the same container constantly, the average RTT in milliseconds is added to the current "weight" of a container which by default is equal to number of currently deployed components times 10. In the case of the marathon reporting service run in the evaluation environment that resulted in placing all the first-stage message processing elements in the producer LAN. The second-stage processing elements were placed either in consumer LAN (in case of LAN 2) or in the producer LAN (in case of LAN 3, connected by the limited-capacity link).

Table 4. Configuration of the REMP instance used for evaluation.

hosts	LAN	REMP instance components
1	1	code repository service worker
2	1	deployment service worker
3	1	component generation service worker
4	1	overlay network manager
5	1	JXTA rendezvous
6	1	producer application
7	1	matching service worker
8	1	consumer applications
9	2	deployment service worker
10-16	2	consumer applications
17	3	deployment service worker
18-20	3	consumer applications

The time observed by a consumer between query registration and creation of the overlay varied typically from 30 to 60 seconds. From the stages required to resolve a query, discovering the query itself by the overlay network manager takes the most (even over 75%)

of processing time. Framework-provided services (including component code generation and compilation, deployment and interlinking of components) do not introduce much overhead.

5.1.4. Evaluation scenarios

This section describes the scenarios in which the overlay network reporting service was evaluated. First, it concentrates on consumer applications, i.e. on their placement in evaluation environment as well as the modes they were run in. Next, the configuration of the Milano City Marathon simulation message producer application is presented.

Configurations of consumer applications

The marathon reporting service underwent experiments of six types:

- “direct 1:1”, in which an application was run in the DIRECT mode,
- “direct 1:5”, in which five applications were run in the DIRECT mode,
- “direct 1:10”, in which ten applications were run in the DIRECT mode,
- “delegate 1:1:1”, in which one application was run in the DEPLOY_NETWORK mode,
- “delegate 1:1:5”, in which one application was run in the DEPLOY_NETWORK mode and additional four instances were run in the REUSE_NETWORK mode,
- “delegate 1:1:10”, in which one application was run in the DEPLOY_NETWORK mode and additional nine instances were run in the REUSE_NETWORK mode.

The experiments were defined also by the applications that were used and placement of particular application instances. Table 5 lists the experiments.

Table 5. The experiments conucted in the evaluation phase (continued on next page).

<table>
<tr><th>Experiment number</th><th>Experiment type</th><th>Host number(s)</th><th>Consumer applications</th></tr>
<tr><td>1</td><td>direct 1:1</td><td>8</td><td>contender fan application, DIRECT mode</td></tr>
<tr><td>2</td><td>direct 1:1</td><td>10</td><td>contender fan application, DIRECT mode</td></tr>
<tr><td>3</td><td>direct 1:1</td><td>18</td><td>contender fan application, DIRECT mode</td></tr>
<tr><td>4*</td><td>direct 1:5</td><td>8,10-13</td><td>journalist applications, DIRECT mode</td></tr>
<tr><td>5</td><td>direct 1:5</td><td>8,10,11,18,19</td><td>journalist applications, DIRECT mode</td></tr>
<tr><td>6</td><td>direct 1:10</td><td>8,10-16,18,19</td><td>journalist applications, DIRECT mode</td></tr>
<tr><td rowspan="2">7</td><td rowspan="2">direct 1:10</td><td>8</td><td>three instances of journalist application, DIRECT mode</td></tr>
<tr><td>10-16</td><td>journalist applications, DIRECT mode</td></tr>
<tr><td>8</td><td>delegate 1:1:1</td><td>8</td><td>professional runner application, DEPLOY_NETWORK mode</td></tr>
<tr><td>9</td><td>delegate 1:1:1</td><td>10</td><td>journalist’s application, DEPLOY_NETWORK mode</td></tr>
<tr><td>10</td><td>delegate 1:1:1</td><td>18</td><td>professional runner application, DEPLOY_NETWORK mode</td></tr>
<tr><td rowspan="2">11</td><td rowspan="2">delegate 1:1:5</td><td>8</td><td>professional runner application, DEPLOY_NETWORK mode</td></tr>
<tr><td>10-13</td><td>professional runner applications, REUSE_NETWORK mode</td></tr>
</table>

Table 5. The experiments conucted in the evaluation phase (continued).

Experiment number	Experiment type	Host number(s)	Consumer applications
12**	delegate 1:1:5	8	professional runner application, DEPLOY_NETWORK mode
		10,11,18,19	professional runner applications, REUSE_NETWORK mode
13**	delegate 1:1:10	8	professional runner application, DEPLOY_NETWORK mode
		10-16,18,19	professional runner applications, REUSE_NETWORK mode
14	delegate 1:1:10	8	three instances of professional runner application, one in the DEPLOY_NETWORK mode, two in the REUSE_NETWORK mode
		10-16	professional runner applications, REUSE_NETWORK mode
15	delegate 1:1:10	8	journalists application, DEPLOY_NETWORK mode
		10	three instances of journalist's application, REUSE_NETWORK mode, different nationality filters
		11-16	journalist's applications, REUSE_NETWORK mode, different nationality filters

* - experiment 4 was also conducted with professional runner's applications
** - experiments 12 and 13 were also conducted with LAN1-LAN3 link bandwidth reduced to 64 kbps

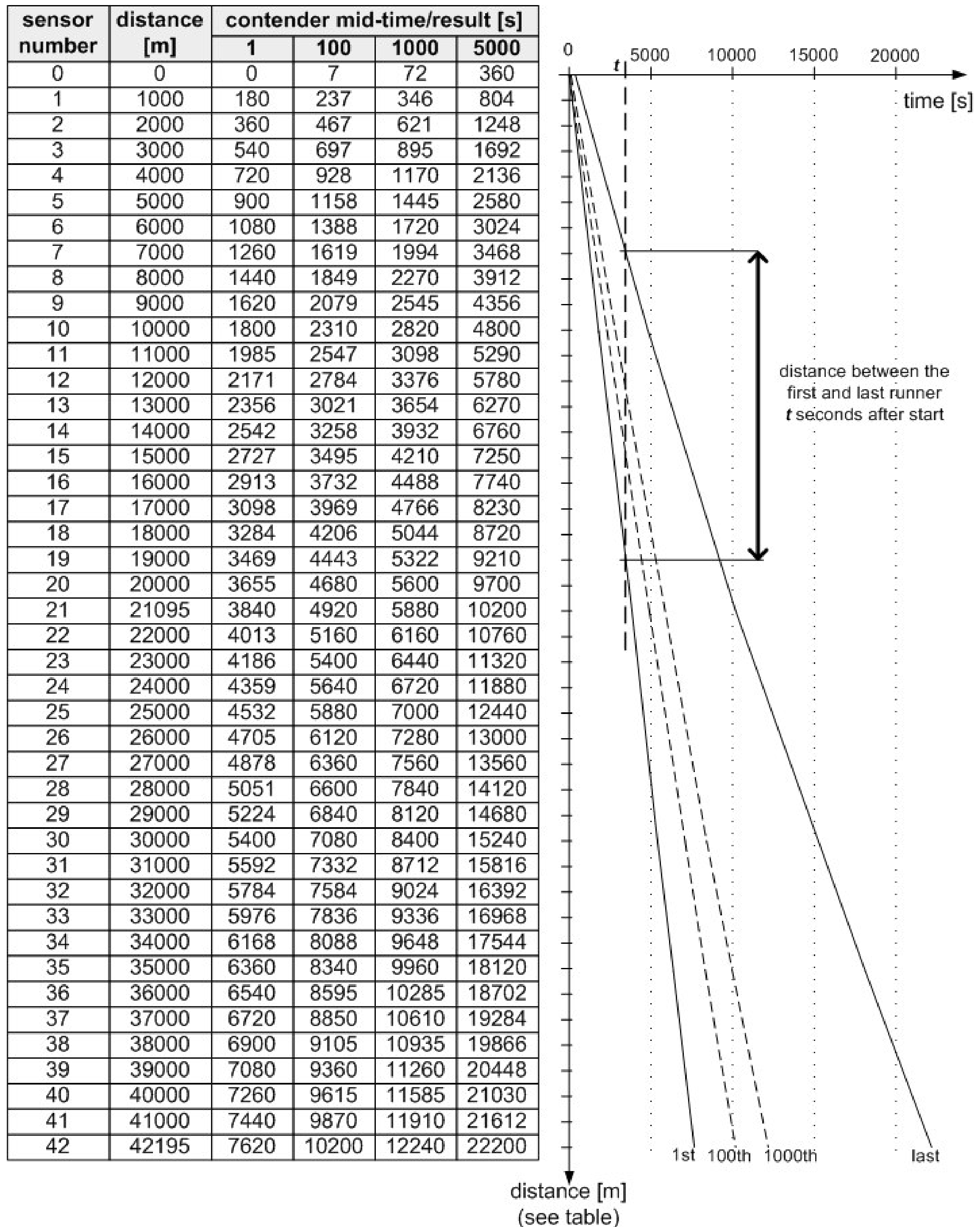
Configuration of the producer application

The message sources were implemented as an application that registers a configurable number of message sources and starts sending messages at a specified period of time. The original time service [MCM 2008] provides reports from only 5 sensors located at 10 km, 21,095 km, 30 km, 35 km and 42,195 km from the race start. Based on the difference between "time" and "real time", times at race start could also be calculated. For 4098 runners, the service needed to produce and distribute 24588 messages in about 6 hours 15 minutes, which was the last contender result. Even for the maximum predicted number of participants (7000), the number of messages would be 42000. Moreover, because the distances between reporting sensors were long, the generated traffic could be bursty, but not voluminous on average. Therefore, for the purpose of the simulation the parameters were changed as follows:

- the number of participants was raised from about 4000 to 5000,
- the number of reporting sensors was raised from 5 (6 including the computed "0" sensor) to 43,
- it was assumed that all of the 5000 contenders finished the race,
- the pace of the last contender was assigned to be equal to the maximum pace allowed by the organizers.

The results achieved by contenders at the added sensors were linearly interpolated on the available data basis.

Fig. 64 illustrates the time-related configuration parameters of message sources. Noteworthy, as the race progresses, the number of active (i.e. publishing messages) sensors raises, because the “length” of the runners pack (i.e. distance between the first and the last contenders) increases. Given that the world class runners are able to cover the whole distance at the same pace and that the maximum pace allowed is also constant, the runners' pack length raises linearly.



sensor number	distance [m]	contender mid-time/result [s]			
		1	100	1000	5000
0	0	0	7	72	360
1	1000	180	237	346	804
2	2000	360	467	621	1248
3	3000	540	697	895	1692
4	4000	720	928	1170	2136
5	5000	900	1158	1445	2580
6	6000	1080	1388	1720	3024
7	7000	1260	1619	1994	3468
8	8000	1440	1849	2270	3912
9	9000	1620	2079	2545	4356
10	10000	1800	2310	2820	4800
11	11000	1985	2547	3098	5290
12	12000	2171	2784	3376	5780
13	13000	2356	3021	3654	6270
14	14000	2542	3258	3932	6760
15	15000	2727	3495	4210	7250
16	16000	2913	3732	4488	7740
17	17000	3098	3969	4766	8230
18	18000	3284	4206	5044	8720
19	19000	3469	4443	5322	9210
20	20000	3655	4680	5600	9700
21	21095	3840	4920	5880	10200
22	22000	4013	5160	6160	10760
23	23000	4186	5400	6440	11320
24	24000	4359	5640	6720	11880
25	25000	4532	5880	7000	12440
26	26000	4705	6120	7280	13000
27	27000	4878	6360	7560	13560
28	28000	5051	6600	7840	14120
29	29000	5224	6840	8120	14680
30	30000	5400	7080	8400	15240
31	31000	5592	7332	8712	15816
32	32000	5784	7584	9024	16392
33	33000	5976	7836	9336	16968
34	34000	6168	8088	9648	17544
35	35000	6360	8340	9960	18120
36	36000	6540	8595	10285	18702
37	37000	6720	8850	10610	19284
38	38000	6900	9105	10935	19866
39	39000	7080	9360	11260	20448
40	40000	7260	9615	11585	21030
41	41000	7440	9870	11910	21612
42	42195	7620	10200	12240	22200

Fig. 64. Configuration of Milano City Marathon simulation case study – raw message sources.

The number of active sensor reaches its maximum about two hours and 8 minutes after the start (Fig. 65), which is the time the winner finishes his run. The first runners having reached the destination, the number of the active sensors starts to be determined by

the distance between the last contender and race finish, rather than between the first and last contenders. Therefore, the number of active sensors decreases and so does the number of reported results. Again, due to the fixed maximum pace, the number of active sensors decreases linearly.

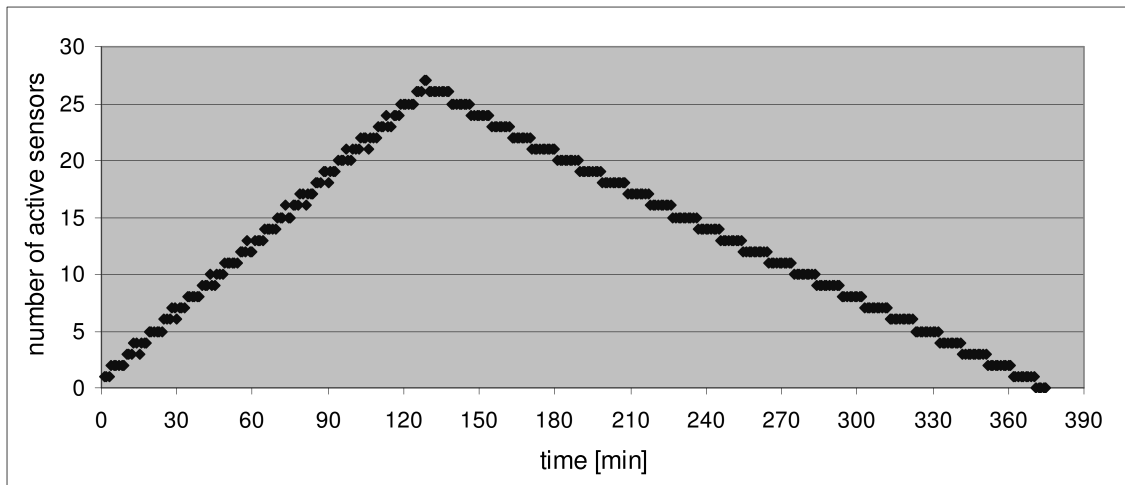


Fig. 65. The number of active (publishing) raw message sources during the marathon results reporting simulation.

The increase in number of sensors (from 6 to 43) results in extending the length of period when the reporting service components will be relatively busy to about three hours (see Fig. 66). Communication channels representing sensors "0" and "1" will be the most stressed ones (Fig. 67), because at the start of the competition the distances between runners are really short, and so are the times between messages.

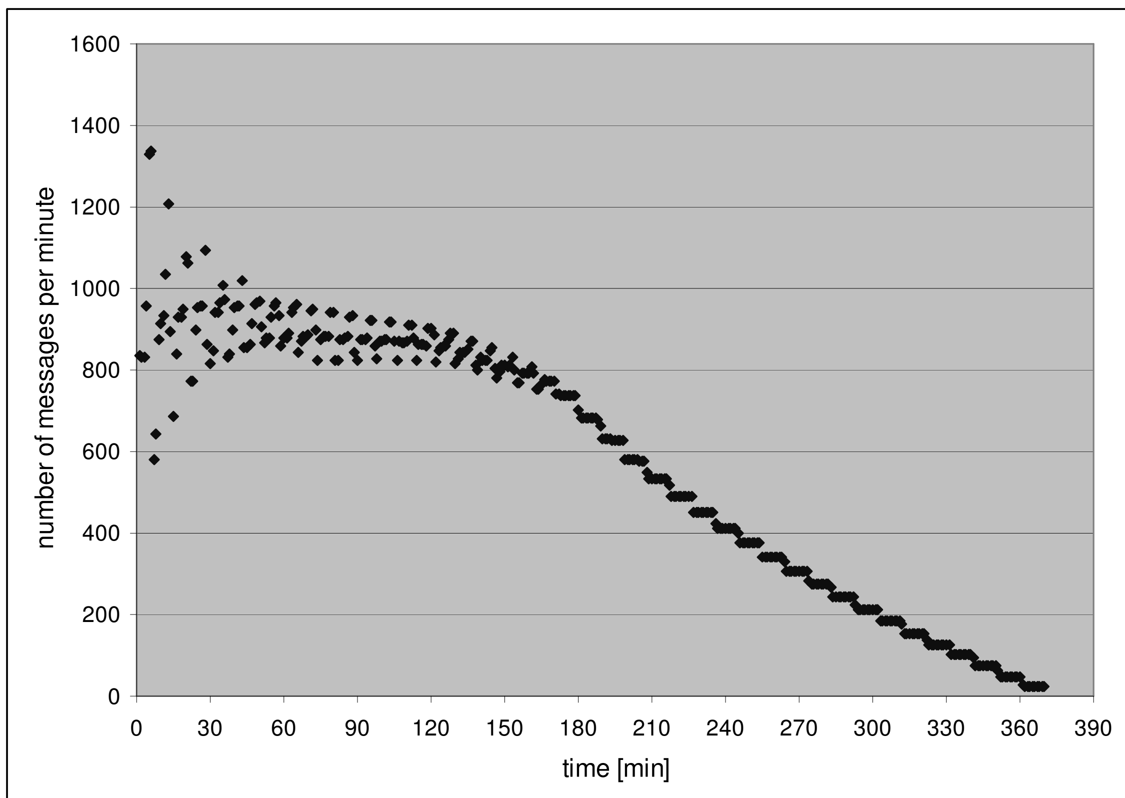


Fig. 66. The expected numbers of messages to be published by raw message sources.

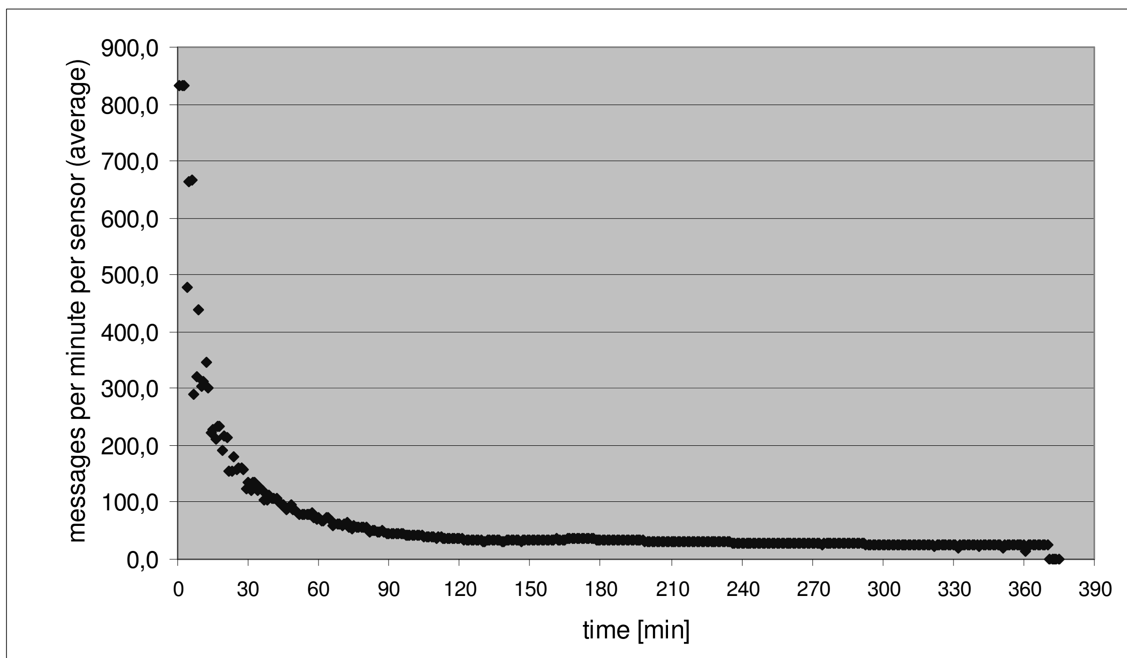


Fig. 67. Average numbers of messages to be published by active message sources.

5.1.5. Evaluation results

This section presents the most interesting results obtained in the evaluation phase. The goal of the marathon reporting service was to enable users of limited (in context of operational memory, CPU and available bandwidth) devices to use the consumer applications. It occurred that because the components attached directly to the raw message sources reduced the amounts of messages greatly, the benefits from delegating the message processing are most obvious when regarding the first processing stage. Therefore, the following subsections concentrate mainly on that components and discuss the reduction of requirements with regard to memory consumption, message loss ratio and CPU usage, respectively.

Effect of distributing consumer application on memory requirements

To illustrate the effect the delegation of processing has on application requirements regarding memory, “direct 1:5”, “delegate 1:1:5” and “delegate 1:1:10” (numbered 4, 11 and 14 in Table 5) experiments were conducted.

Fig. 68 illustrates the heap size of the professional runner consumer run in DIRECT mode (i.e. gathering data directly from message sources), which is, at its maximum, over 200MB.

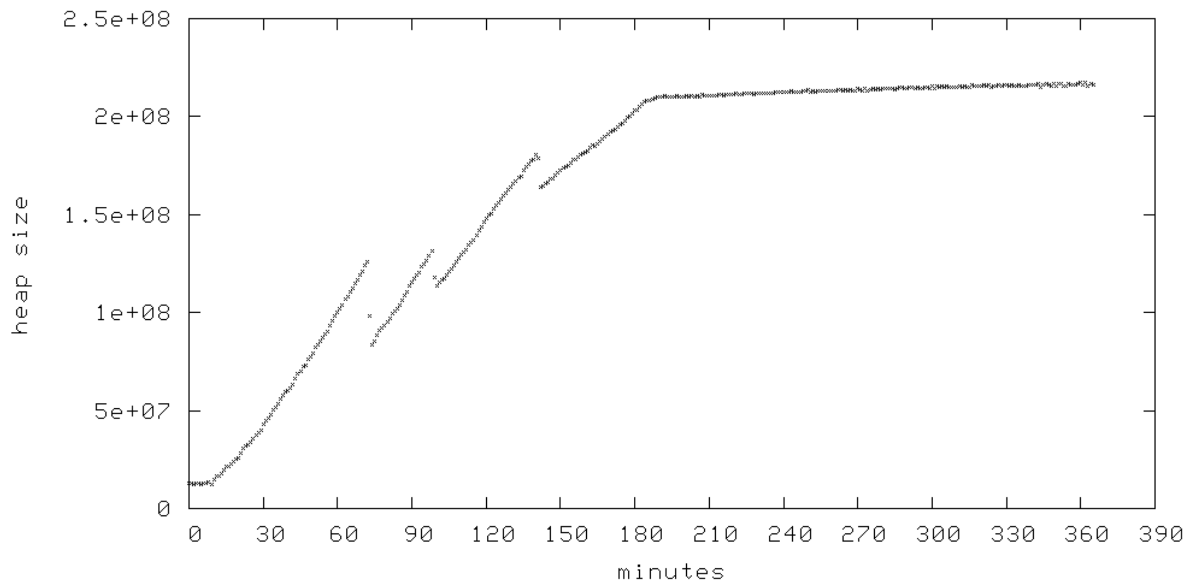


Fig. 68. Directly connected consumer heap size
(experiment 4).

Fig. 69 and Fig. 70 illustrate the same application distributed between consumer application, which reports only the “interesting” messages to its user and middleware-provided deployment service worker, that does most of the processing and all logging. The maximum heap sizes are at about 16MB and 210MB, respectively.

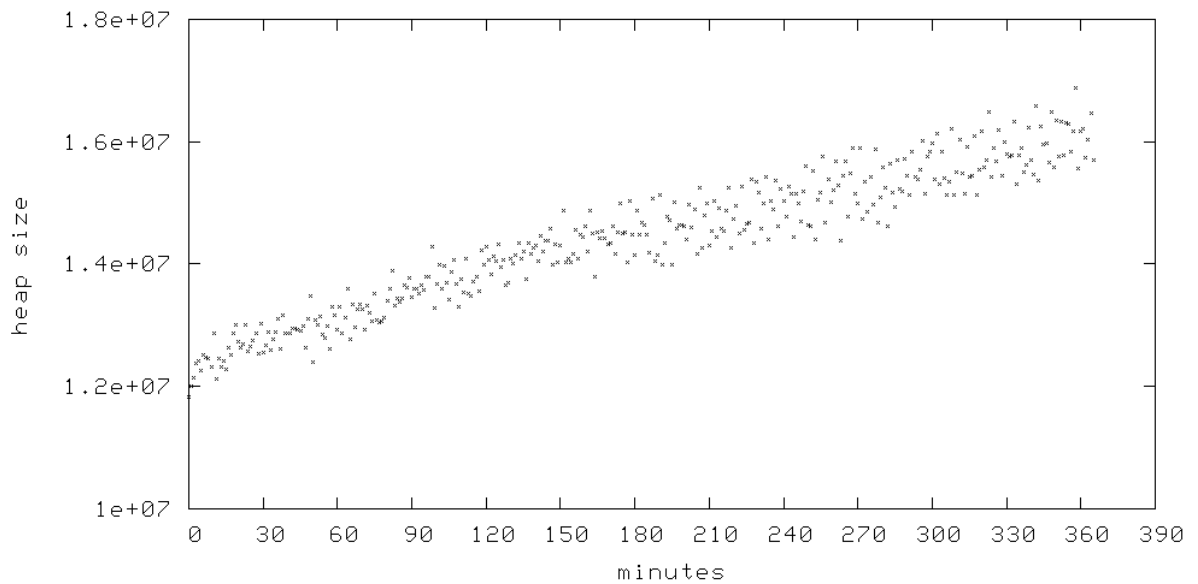


Fig. 69. Professional runner consumer’s user interface heap size
(experiment 11).

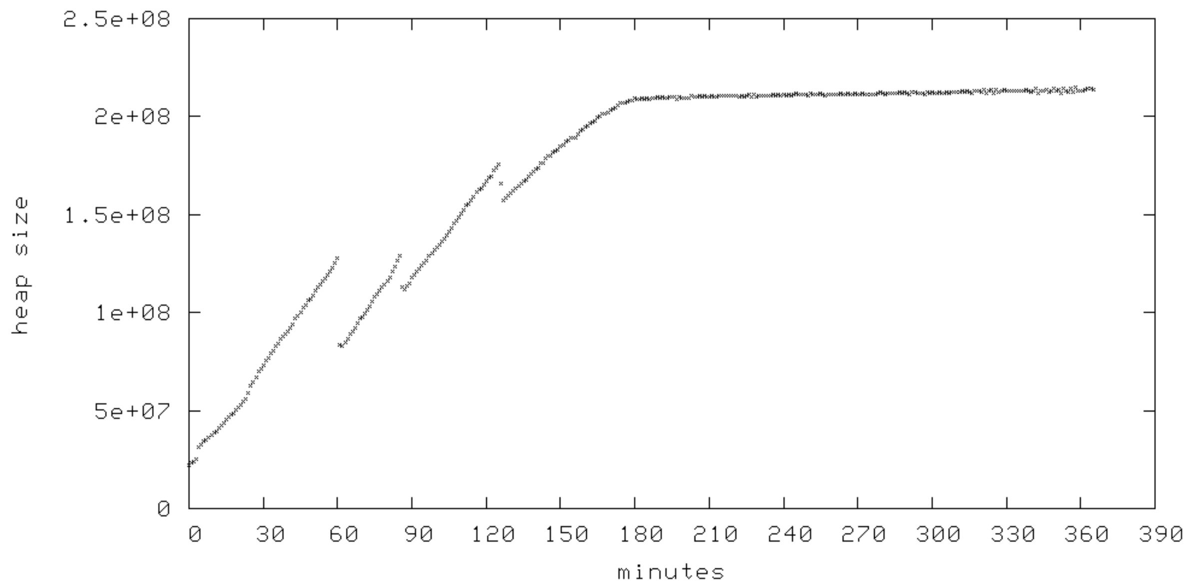


Fig. 70. Deployment service worker heap size
(experiment 14, "top 10" filter and average pace counter deployed).

Of course, in case of one client deploying its private overlay network, the sum of memory requirements is higher than when the application was not distributed. However, if the network is constructed with reuse in mind, attaching a new consumer does not result in severe growth of memory requirements. In fact, Fig. 70 illustrates the memory consumption of a deployment service worker hosting all the processing components, that were reused by 10 consumers. In such case, the reuse leads to lowering the overall memory consumption greatly. The store of messages received by the user interface components uses about 2MB of memory, thus making it possible to execute the professional runner application on a very limited device.

Effects of distributing consumer application on message loss

The peers that were used in the experiments were configured to use (unreliable) propagate pipes constructed over reliable TCP transport. Therefore, the decision of whether to drop a message or not, was taken by JXTA, not by the transport protocol. As a baseline, a series of "direct 1:1" experiments (numbered 1 and 2 in Table 5) were conducted. The tests yielded an average loss ratio of 0,02%. The numbers of messages received by the consumer (Fig. 71) were very close to the ideal case (Fig. 66). Although at start, presumably due to communication channels setup, the messages arrived out of order, after 60-90 seconds the order was restored. Most of lost messages were sent out by sensor "0", therefore making it impossible to calculate race result for 2 participants, on average.

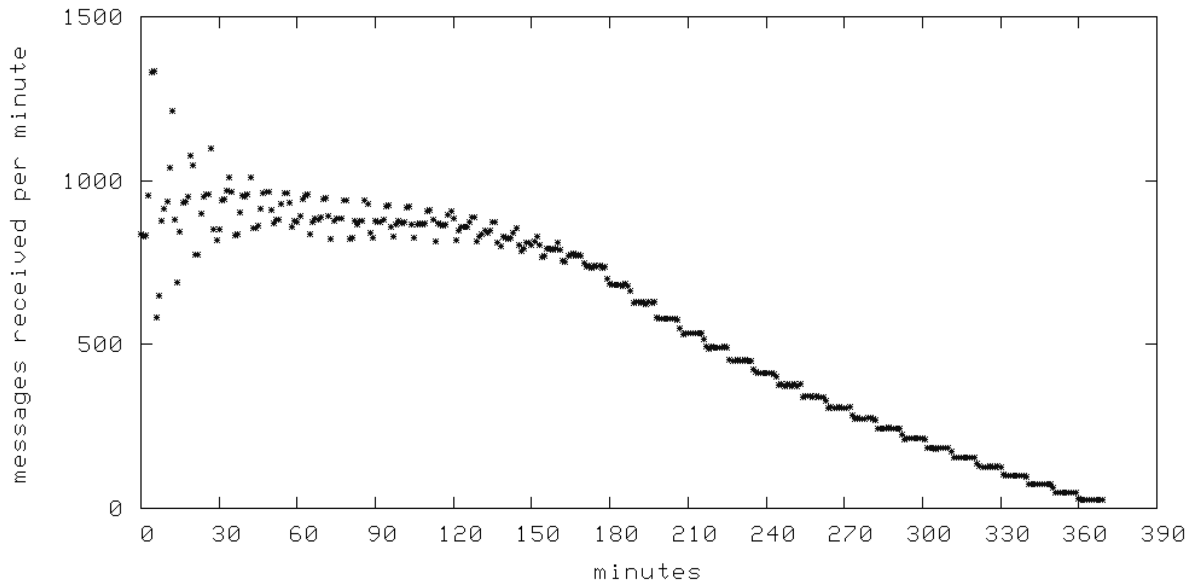


Fig. 71. The numbers of messages received by a consumer in a "direct 1:1" scenario (experiment 2).

The next series of tests was conducted using five journalists application consumers run in DIRECT mode, one of them at the raw message sources LAN and four attached by a single 4Mbps WAN link (experiment 4 in Table 5). Remote consumers experienced 0,7% message loss, with most of lost messages sourced at sensor "0". Taking into account that the lost messages made it impossible to calculate race results of even 100 (out of 5000) participants, the loss is significant.

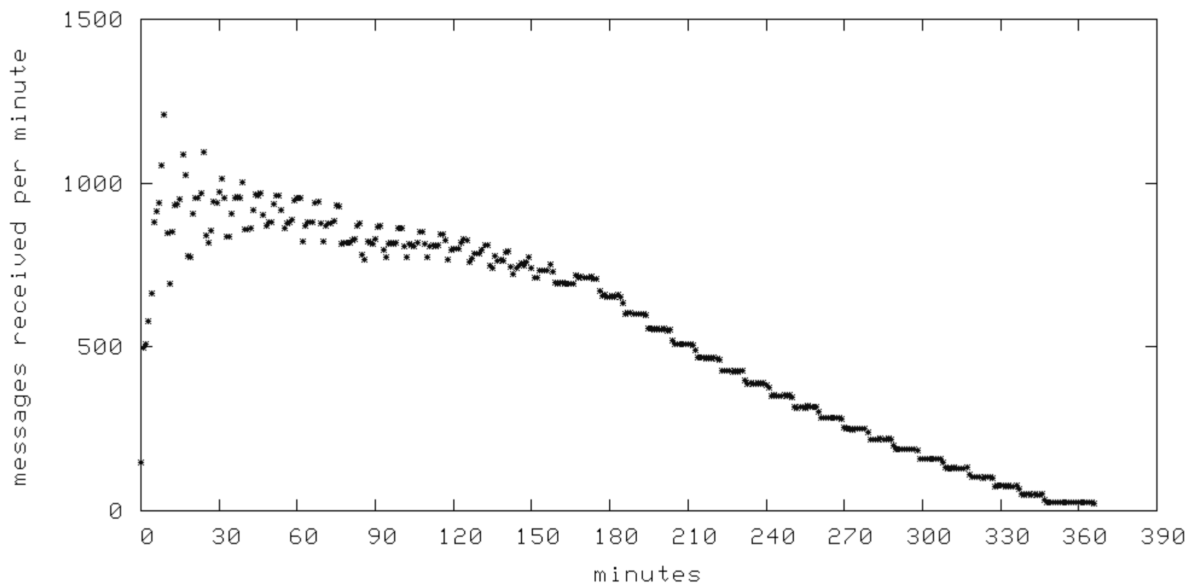


Fig. 72. The numbers of messages received by a consumer connected through a lower speed link in a "direct 1:5" scenario (experiment 4).

Another series of tests were conducted using 10 receivers attached directly to the raw message sources (experiment 6 in Table 5). One of them was present in the raw sources LAN, seven connected with 4Mbps WAN link and two used a more restricted, 640kbps WAN link. The consumers were struck by a large message loss rate, which was even 2,6% in case of the 4Mbps link and over 7% in case of the 640kbps link. In case of the restricted link it happened that some of the communication channels failed to set up due to communication

timeouts (there were 86 communication channels required by the peers on that network set up in parallel). Such case is illustrated by Fig. 73.

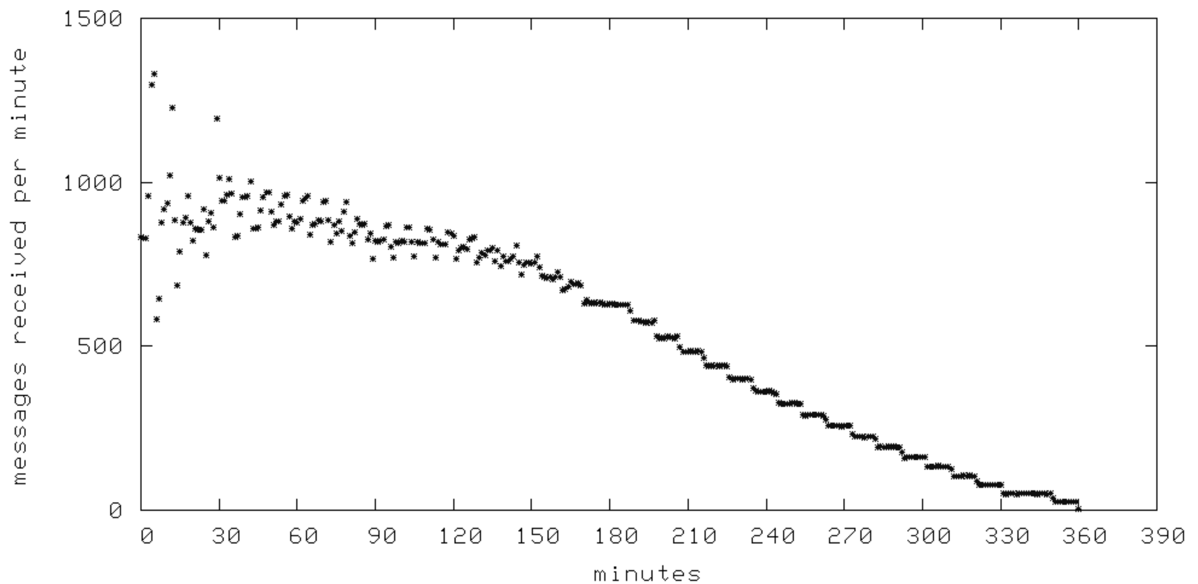


Fig. 73. The numbers of messages received by a consumer connected through a restricted link in a “direct 1:10” experiment (communication channel from message source “43” failed to set up).

A closer look on the sender and consumer logs reveals that for over two hours the received messages arrived out of order and the time intervals between them varied greatly. Most messages that were dropped were sent either during first 30 minutes of operation or between 70th and 90th minute. Fig. 74 illustrates the rate the message arrived to the consumer through the congested link. Observation of sender CPU usage (later in this section) shows that during the congestion periods the usage was significantly lower, with some of the channels just waiting on acknowledgements and buffering messages, some of which were eventually dropped.

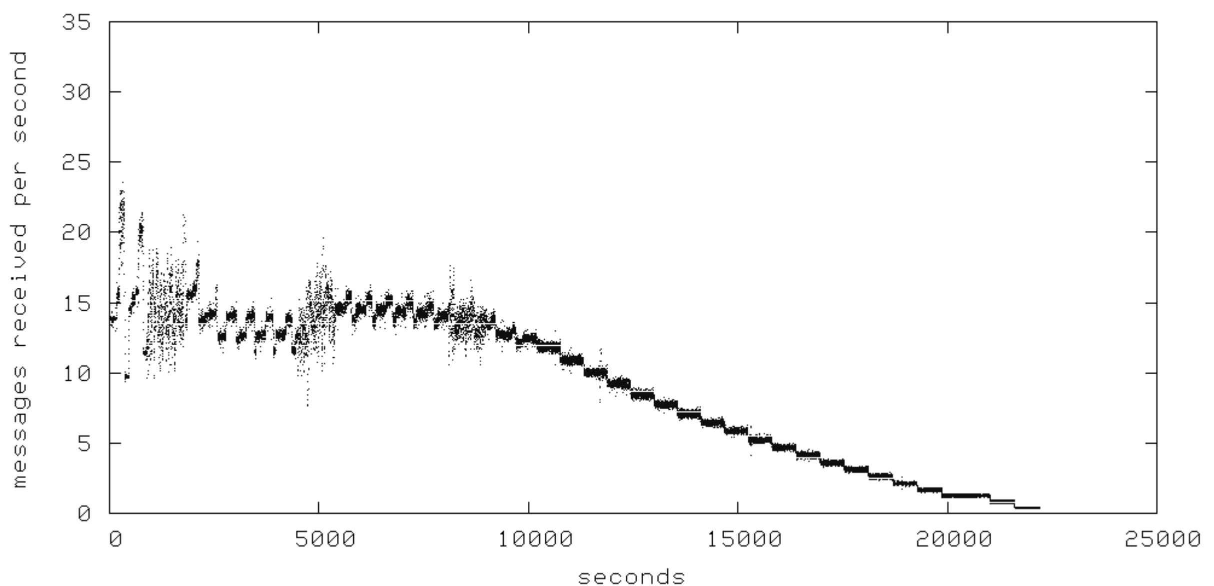


Fig. 74. The numbers of messages received by one of professional runner consumers connected through the 640kbps link.

To check whether delegation of processing solves the problem, similar tests were conducted. Instead of connecting directly to the raw message sources, the consumer

applications were configured so that one of them deploys a new overlay network, which is then reused by others (experiment 13 in Table 5). Fig. 75 illustrates the numbers of messages received by an application connected through the same 640kbps link that reused the overlay. Because the message processing components were placed in the raw message sources' LAN, no message loss occurred, although at simulation startup, some arrived out of order.

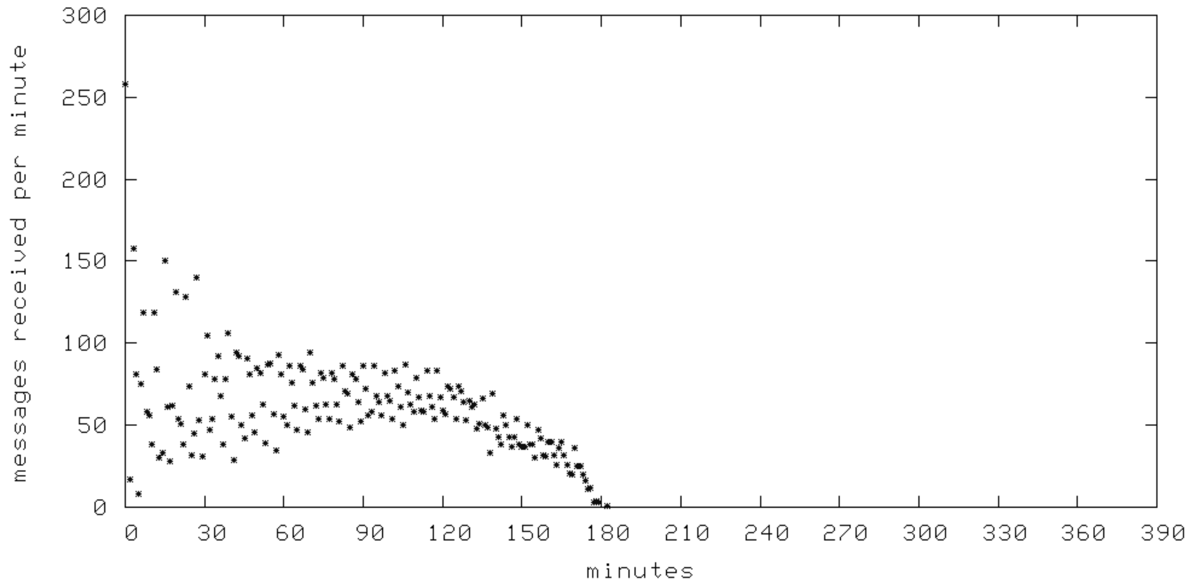


Fig. 75. Numbers of messages received by a professional runner client application connected through the 640 kbps link.

As expected, delegation of components that reduce the number of messages transmitted to the consumer resulted in reduced bandwidth consumption, making even low bandwidth links sufficient for the application. The two users that shared the 640 kbps link did not suffer from congestion. Experiment 13 was repeated with the low-capacity link bandwidth reduced to 64kbps. In the first 30 minutes the clients suffered a from significant message loss and out of order delivery. After that time, even the very restricted link provided reliable service. Taking into account that first 10 kilometers' pace seem to have a lower relevance for a professional runner than the later 32, by delegating the message processing to the overlay network, two professional runner applications can provide satisfactory service to their users even in case of sharing a low-capacity, 64kbps link.

Effects of distributing consumer application on CPU usage

Message sending is the main task of the producer application. The operations on message contents (calculating time differences and averages) are very simple and do not take much processor time. Fig. 76 illustrates the baseline – CPU utilization of a producer application that registered 43 message sources, each of which publishes to one receiver (experiment 3 from Table 5).

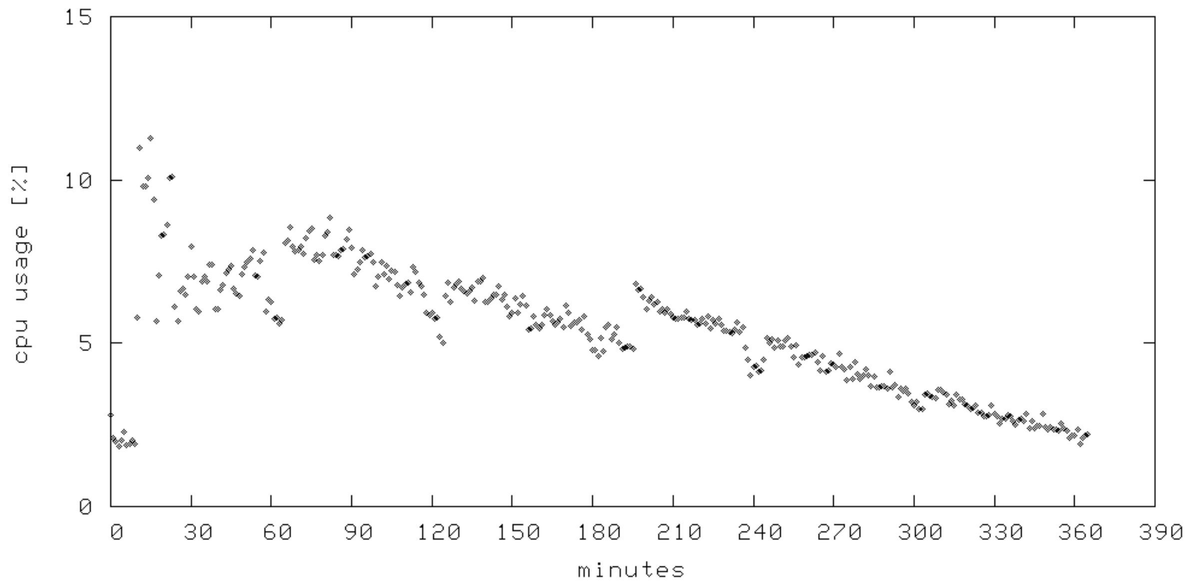


Fig. 76. Producer application CPU usage (experiment 3 - “direct 1:1” scenario).

With the number of message receivers, CPU usage increases greatly. Fig. 77 depicts CPU usage for the same message sources serving 5 receivers (experiment 5 in Table 5).

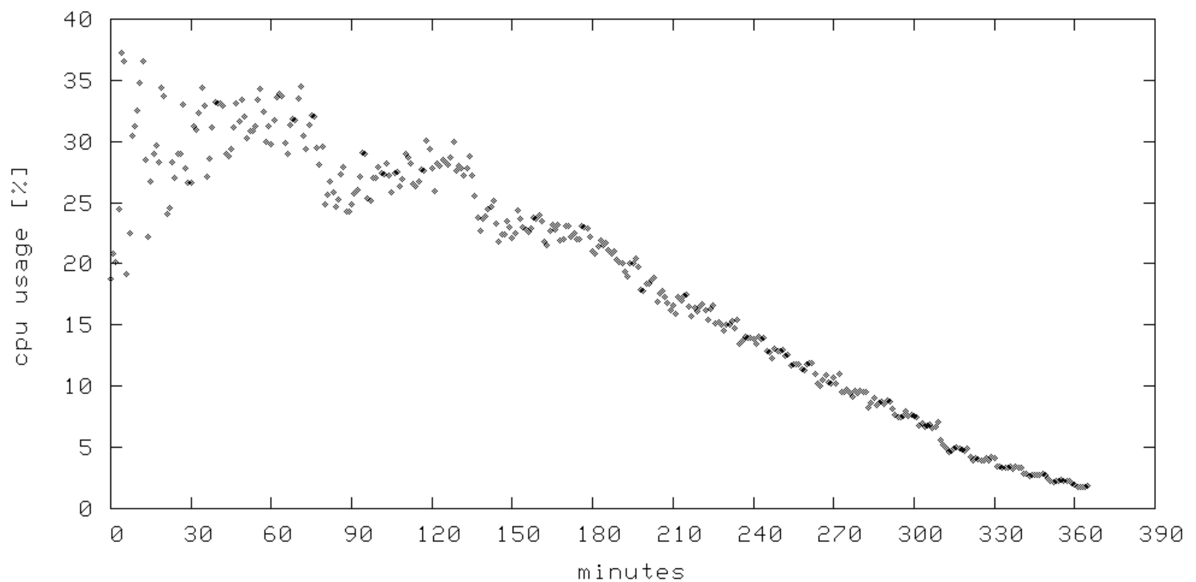


Fig. 77. Producer application CPU usage (experiment 5 - “direct 1:5” scenario).

10-12 receivers, (i.e. 430-516 communication channels, or about 15000 messages per minute) seem to be the maximum capacity of the publisher host used for the test (see Fig. 78). Obviously, that cannot satisfy the requirements of Internet-wide marathon reporting service. Even if the sources were implemented as separate applications, the number would not be much greater, because at the peak, which occurs just after race start, only two sensors are publishing messages (see Fig. 66 and Fig. 67). On the other hand, if only the consumers are not allowed to connect directly to the message sources, that means that a single, low-end machine can serve up to 10-12 first stage message processing components, which seems to be enough for the marathon reporting service.

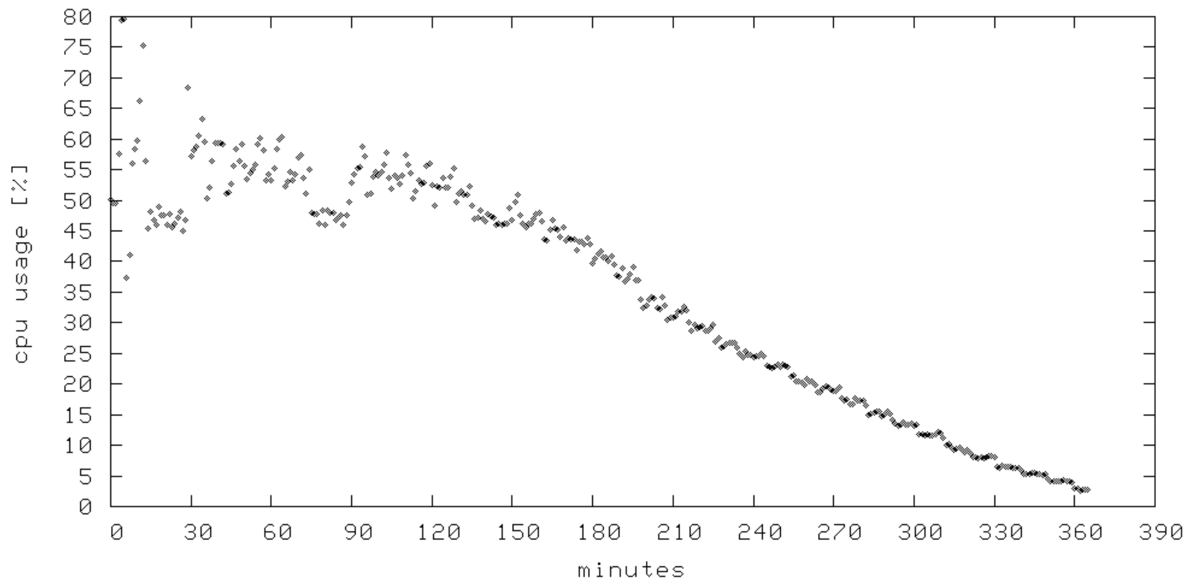


Fig. 78. Producer application CPU usage (experiment 6 - “direct 1:10” scenario).

The 1:10 scenario experiments were conducted in a configuration described earlier, which included two receivers connected with low-bandwidth, 640kbps link. Two periods of congestion were observed, roughly between 10-30 minutes and 70-90 minutes after the start of experiment. Note that the CPU utilization graph shows lower values in these periods, thus ensuring the congestion took place at the link, not at the sender machine.

On the other communication end, in a naive “direct” approach, the consumer must process everything the producers publish, so the CPU load figure (Fig. 79) resembles the one that illustrated publisher’s load in 1:1 scenario.

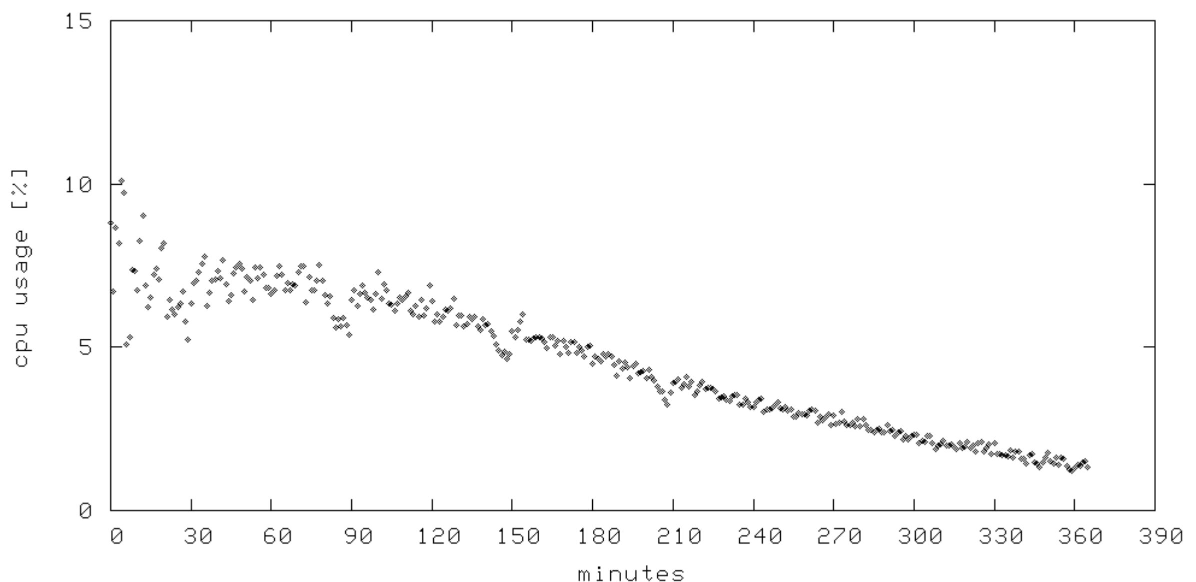


Fig. 79. Consumer application CPU usage (experiment 6 – “direct 1:10” scenario).

Delegation and reuse of message processing allows to limit the number of components that need to receive all raw messages. Therefore, in the case of two components (“top 10 filter” and “average counter”) deployed the CPU load of the publisher is reasonably low (Fig. 80).

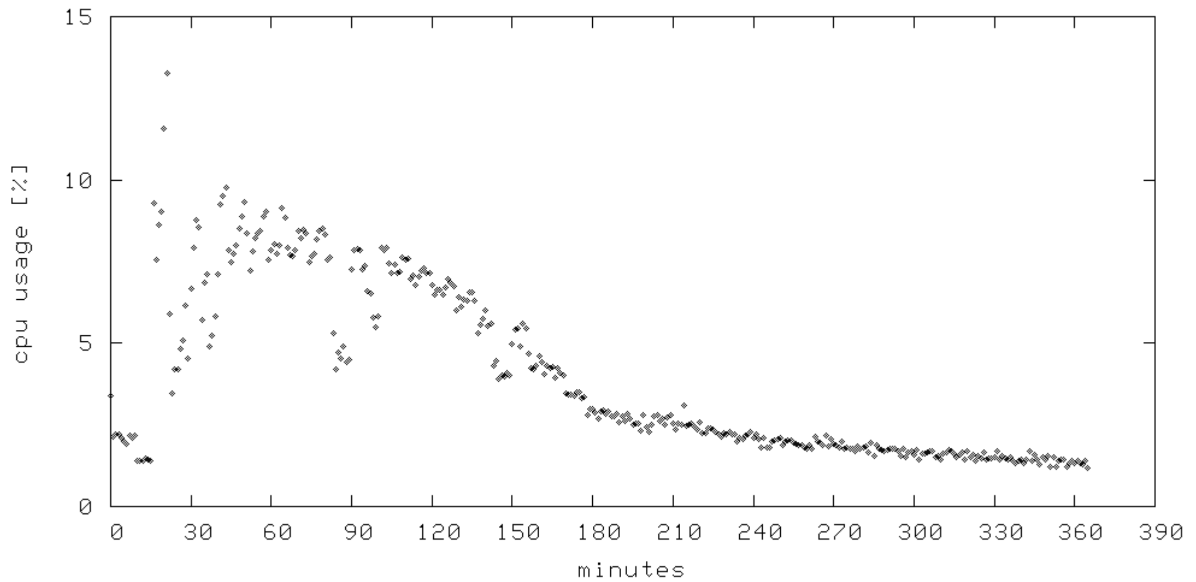


Fig. 80. Producer application CPU usage (experiment 8 - two processing components attached).

Because the first stage processing components reduce the number of published messages greatly (in the Milano City Marathon case from 215 000 to less than 11 000), the loads introduced by subsequent stages of processing are relatively low. Fig. 81 shows the load of a deployment service worker hosting first-stage processing component that serve one "nationality filter" hosted at another machine. Fig. 82 shows similar figure for 10 second-stage processors, respectively.

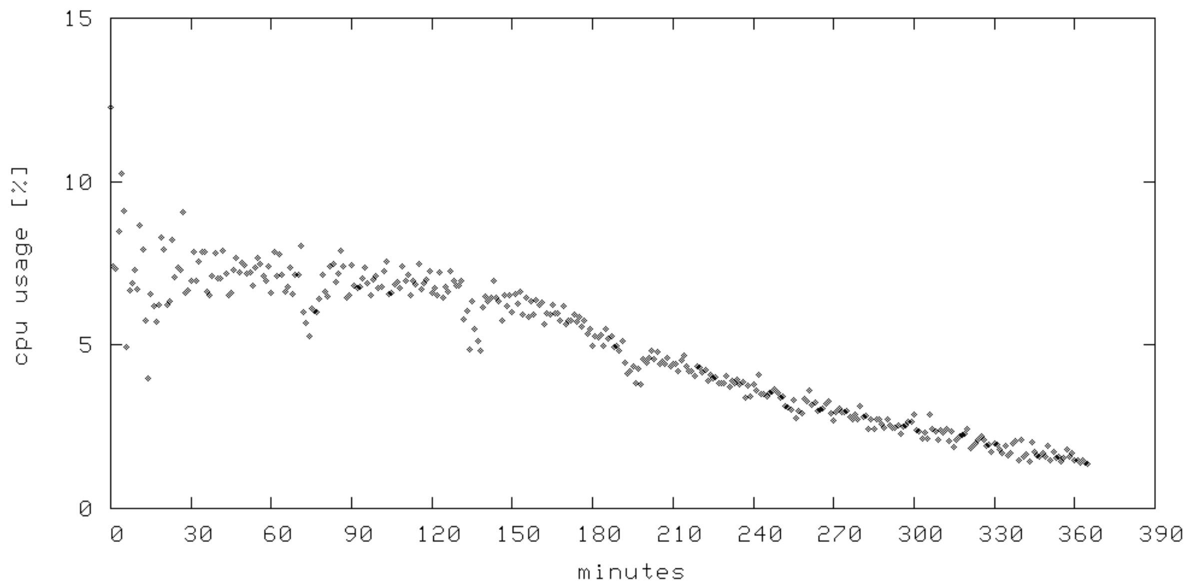


Fig. 81. Deployment service worker CPU usage
(experiment 9 - one processing components deployed, 1 receiver attached).

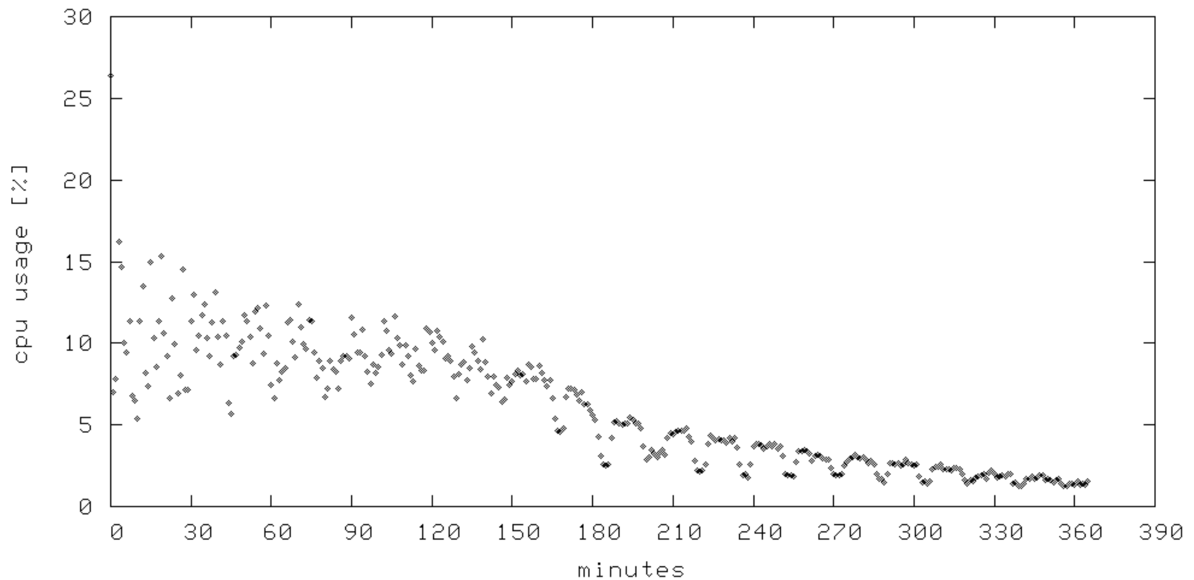


Fig. 82. Deployment service worker CPU usage
(experiment 15 – one processing component deployed, 10 receivers attached).

In the analyzed scenario, the second-stage processors introduce very little overhead to their containers. Fig. 83 illustrates the CPU utilization of a deployment service worker application hosting one second-stage message processing component.

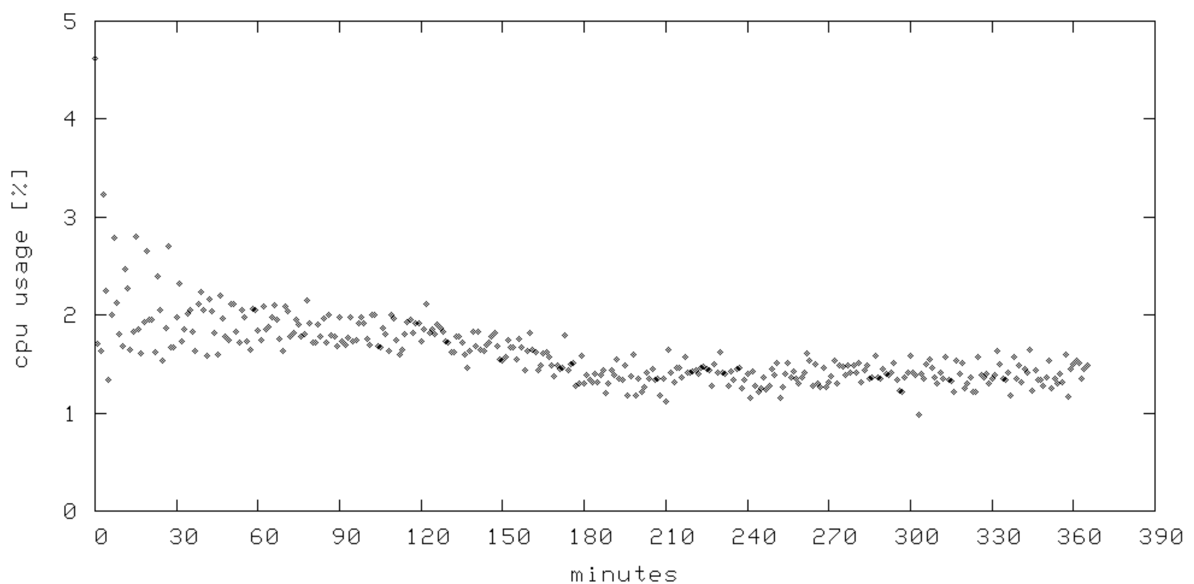


Fig. 83. Second-stage deployment service worker CPU usage (experiment 9 - 1 nationality filter deployed).

Experiments described in this section suggest that not only the end user applications, but selected message processing components could be hosted on devices limited in terms of CPU power, if only there was a guarantee the devices stay connected to the network.

Conclusions

This section concentrated upon a messaging intensive use case. Noteworthy, the message streams carried much unnecessary information that could be simply filtered out close to the source. Furthermore, the processing algorithms could be easily split into separate components interoperating asynchronously. That allowed to demonstrate how an application, by delegating some of its functionality, may benefit on CPU utilization, memory

and bandwidth consumption, while not sacrificing any of its features. The delegation allows to run quite complex distributed applications with limited user machine with a low bandwidth network connection.

Fig. 84 illustrates the effect of message processing delegation on message consumer end user machine memory consumption. As shown earlier in this section, in case of the presented case study delegation of message processing resulted in “delegation” of memory consumption, thus allowing the user to run the application on a device with limited amount of memory. Moreover, the results of computations performed by the delegated components, thanks to REMP mechanisms, could be reused much easier, than the results of computations performed by the end user application.

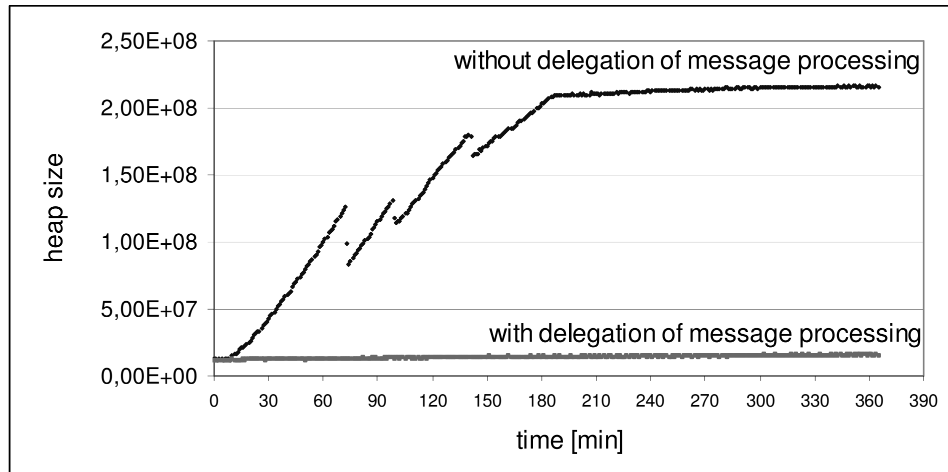


Fig. 84. The effect of delegation of message processing on message consumer’s memory consumption.

Fig. 85 presents the gains in CPU utilization on the producer’s side. In a messaging intensive scenario, such as in the case of the presented case study, the CPU utilization at the message producer’s side depends mainly on the number of messages to be marshaled and sent over the network. If the consumers are designed to reuse the intermediary computation results, the number of connections to raw message sources is lowered and so is the CPU load of the producer.

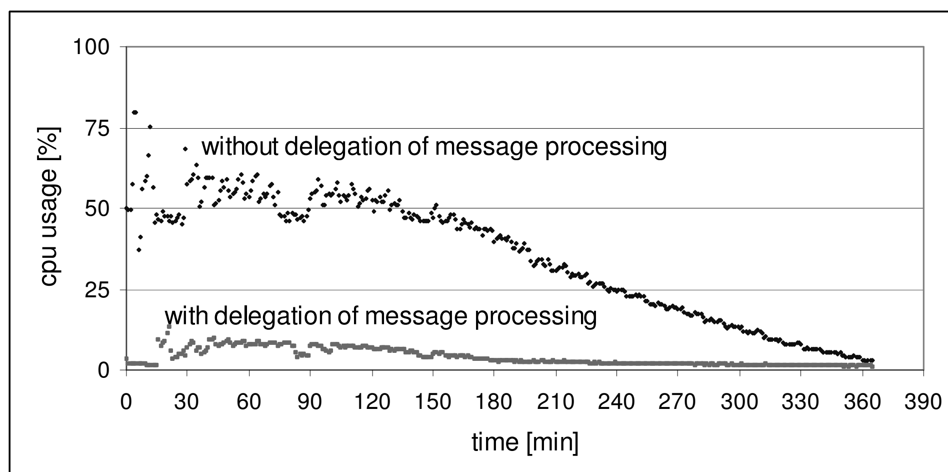


Fig. 85. The effect of delegation of message processing on message producer’s CPU utilization.

Fig. 86 illustrates the number of messages received by the end user application with and without delegation of the processing. The number of messages is closely related to the

bandwidth requirements of the applications. In that way, Fig. 86 illustrates the gains from delegation of message processing nodes in terms of bandwidth consumption.

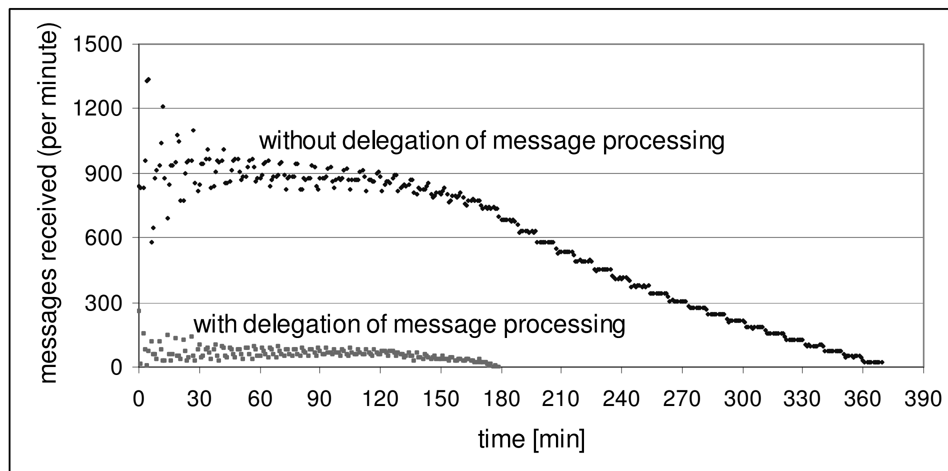


Fig. 86. The effect of delegation of message processing
on the number of messages received by end user's application.

Even simple overlay networks constructed by REMP prototype take a minute or two to set up, primarily due to latency introduced by the reused JXTA Discovery Service. That makes REMP a poor choice for creating networks that are expected to last only for a few minutes.

To summarize, the use case showed that the prototype provides a good base for creating messaging-intensive, long-lasting overlay networks. Provided that the message stream carries much irrelevant information, the prototype offers support for end-user devices that are limited in terms of operational memory, CPU power and available bandwidth.

5.2. Examples of other applications

In order to describe the spectrum o possible framework applications in more detail, this section presents two possible use cases that could be implemented either with the framework prototype or with the fully functional framework: classic stock monitoring example and a more hypothetical accident rescue system.

5.2.1. Stock monitoring system

Monitoring stock market is a classic example used when describing operation of publish/subscribe systems. In a typical scenario, an end user wants the system either to report the prices of shares of interest continuously or to raise an alarm when something unusual, like sudden price change, occurs. Typically, such unusual condition is detected either when the price of shares falls below or raises above a specified threshold expressed either in monetary units or price percentage counted over a specified period of time. Stock brokers frequently offer their clients automated buy or sales operations if such changes are detected.

From the REMP framework point of view, the stock monitoring example is different than the marathon reporting service mainly because the message producers may be geographically separated. Therefore, the service effectiveness would be determined mainly by proper choice of deployment service workers to be used. The default strategy based on

number of components and RTT measurements seems to be too simple to host many processing components, because the volumes of distinct message streams could vary greatly, also in time domain – markets do not operate on a 24 hours per day basis and their operational hours are different worldwide (consider time of the day difference between Tokyo and London). Therefore, the stock monitoring system could benefit not only from dynamic deployment, but also from migration of processing nodes in order to move the processing nodes closer to the more "active" message source. Because in the REMP prototype migration could be initiated only manually, the prototype support for a stock monitoring system is limited. Nonetheless, implementation of such system is still possible.

The ways of defining conditions for detecting important changes are quite simple and not flexible, being based either on single valor or a stock index calculated for a wide range of shares. A regular (not VIP) client of a stock brokering office usually is not able to define his own index based upon shares that are of particular interest to him. Moreover, the user might be interested in tracing price changes on different markets, e.g. trading industrial resources, such as oil and use more than one stock service.

As an example of a set of rules specified by an end user, consider:

```
RuleSetA: evaluation_period(1 hour) {
  if( price_change(marketA, oil, last_week)<-5% ) {
    sell(market, oilCompA, 50%);
    set(ruleSetB);
  }
  if( value_change(oilCompIndex(), last_day)<-3%) {
    sell(marketB, oilCompA, 100%);
  }
  if( value_change(oilCompIndex(), last_day)>1% ) {
    buy( marketB, oilCompA, 20000PLN );
  }
}

RuleSetB: evaluation_period(10 minutes) {
  if( price_change(marketA, oil, last_day)<-2% ) {
    sell(marketB, oilCompA, 100%);
  }
  if( price_change(marketA, oil, last_week)>2% ) {
    buy( marketB, oilCompA, 10000PLN );
    set(RuleSetA);
  }
  if( value_change( oilCompIndex(), last_week )>3% ) {
    buy( marketB, oilCompA, 20000PLN );
  }
}

oilCompIndex() {
  return
    0.1*price(marketB, oilCompA)+
    0.3*price(marketC, oilCompB)+
    0.6*price(marketC, oilCompC);
}
```

Listing 12. A simple set of rules for automated stock market application.

When analyzing even the simple set of rules, several user requirements may be observed:

- the user wants the system calculate an index based on prices of shares of different companies,

- the prices should be obtained from different stock markets, i.e. the message streams provided by different markets are expected to be merged at some point in the network,
- the system should keep a history of traced values, i.e. the virtual message producer connected to the message sink should be permanently active or be able to save its data between activations,
- the executed rule sets may be different and evaluated more or less frequently.

Such set of conditions may be served well by a quite simple, static structure built from objects that come as a result of translating the rules to a computer-understandable form. However, in more complex scenario, e.g. when the user who wants to use various oilCompIndex() functions depending on the situation on the market, the static structure would either waste resources by reporting irrelevant data or keeping unnecessary objects active and waiting for the conditions to change. Taking into consideration that stock monitoring systems must deal with huge number of price reports, the ability for early filtering out unimportant information becomes a crucial requirement in context of system scalability.

The proposed framework services support creation of such applications in several aspects:

- the programming language for components that perform calculations is very flexible,
- introduction of a new programming language for components requires the developer only to provide component generation service workers (and maybe deployment service workers),
- existing containers could be integrated with the framework by making them implement a deployment service worker interface,
- introduction of a suitable deployment strategy does not require the programmer to re-implement any part of the deployment service,
- the monitoring and metering service is easily extensible – that provides for easy introduction of new runtime metrics.

In case of financial systems, security is often a crucial aspect. Although the framework does not introduce its own security mechanisms, it reuses the underlying JXTA middleware, which is capable of providing secure communication channels and certificate-based authentication of peers that join the network. The communication channels could be used by overlay network designer just by setting a flag in the link specification, while using a certificate-based peer group membership service is a bit more complicated and not supported by the prototype.

5.2.2. Accident rescue system

As an example use case of the fully functional framework consider a street monitoring system enhanced to report and organize rescue teams to help the victims of possible accidents. In order to be properly managed, the accident must be reported by a connected entity, e.g. by a traffic sensor, a sensor installed in a car, or a person that notices the event. Assuming that the signal carries information only about the location the accident happened, in order to decide what means should be mobilized to help in rescue action, the system must obtain more detailed information about the emergency. To do that, it must search for e.g. a street monitoring camera located in proximity of the accident location and connect it to a video stream analyzing software. The analyzer would be used to determine what exactly happened. Based on the results of the image analysis, the system should look e.g. for nearest doctor, fire brigade, police car, etc. and ask or order them to go to the

emergency area. At that stage, the emergency team would grow and contain not only computers but humans as well. Note that additional actions may be also taken, e.g. the video stream obtained from the street monitoring can be useful for the firemen and therefore should be sent to them when they join the rescue action.

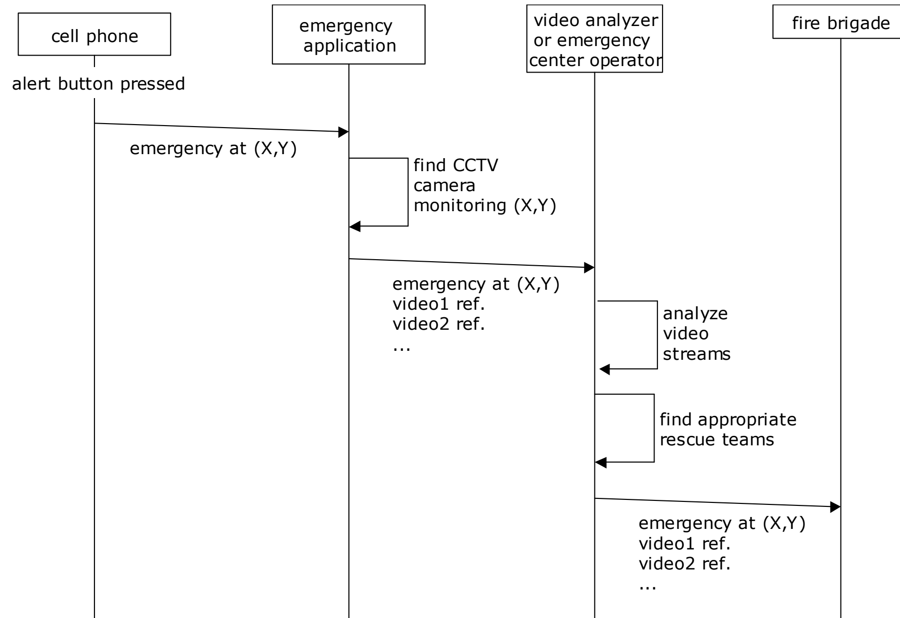


Fig. 87. The hypothetical flow of accident reporting messages.

The scenario sketched in the previous paragraph results in creating a mission-specific group of common interest, that could be interconnected using a dynamically created overlay network. It is up to the accident rescue system to select the set of group members. A variety of complex conditions that could be used to select the members might be proposed, e.g. "select an entity that represents a doctor that is near the place where the accident happened", that in real life are expressed just like in the example, in concept-based way. Note also the – typical in critical situations – use of words "an" and "a", that in real life mean anycast addressing.

From the framework instance point of view, at least the following entities may be identified:

- message sources: alert devices, emergency teams equipment (e.g. GPS-enabled devices),
- message sinks and processing components: emergency applications (either automated or not).

When not in real use, the system would be required only to keep track of available resources by using discovery and monitoring services. Because the communication endpoints are known only at runtime, the main task for the middleware would be to interconnect the entities on demand. Note that because messages of the same meaning may be sent by various devices using various communication protocols, the choice of an Internet-enabled peer-to-peer execution environment seems to be reasonable. Because the communication channels may be created and destroyed very frequently (consider a cell phone inside a car going through a small monitored area), monitoring the presence of an entity e.g. by using the monitoring and metering service, is crucial.

On the other hand, the scenario leaves little place for component code generation and compilation. The dynamic deployment could be of use, i.e. for uploading a piece of software e.g. to a device installed in a fire truck. The components, however, would rather be

pre-compiled and stored in the code repository service earlier than at network creation time.

5.3. Summary

The evaluation of the marathon reporting use case described in this chapter shows that even the not fully functional prototype provides features that support creation of a range of interesting applications. The virtues of the proposed approach are visible especially when considering running applications requesting large numbers of messages with low capacity devices. In the use case, delegation of message processing to the middleware and reuse of the deployed components resulted in saving on bandwidth, memory and CPU utilization.

The other described use cases, although not implemented, show that the area of framework applicability is not limited to feeding the consumers with recent sport results. The classic example of stock monitoring application shows that the systems implemented with the proposed framework could also cover an area commonly assigned to message broker-based systems (in the “worst” case, the framework could run such brokers as dynamically deployed components). The hypothetical accident rescue system would benefit from the inherited framework’s ability to interconnect various devices in an uniform way. By adding semantic descriptions and concept-based queries regarding the sources, the framework increases the expressiveness of entity selection and therefore could make creating the mission-oriented groups easier.

6. Conclusions

The presented work was focused on the architecture and partial implementation of a generic framework for deploying component-based applications oriented on message processing in peer-to-peer networks. The framework extends an existing generic-purpose middleware with features that open new possibilities to application developers. This chapter summarizes the presented work in context of the thesis and research goals specified in section 1.4.

The chapter is split into two sections. Section 6.1 presents the most important achievements of the presented work, while section 6.2 summarizes its deficiencies and specifies the most important areas of future development.

6.1. Research achievements

The thesis (defined in section 1.4) was focused on two aspects: building a framework for dynamic deployment and introduction of semantic message descriptions. Therefore, this section focuses mainly on the achievements related to the aspects (in subsections labeled "TA1" and "TA2", respectively). Furthermore, the same section specified four research goals to be achieved by the framework. This section briefly refers to each of them also (in subsections labeled "RG1" – "RG4").

<u>TA1. Dynamic deployment</u>

The first part of the thesis was proven by the implementation of a working and stable REMP framework prototype. The author of the thesis run several series of experiments upon the prototype, all of them successful with regard to the "*dynamic deployment of message processing overlay networks*" part. Therefore, the services designed and implemented as a part of the thesis, proved to be usable and working.

In order to provide a flexible service, the framework introduced and partially implemented a series of novel concepts, overviewed in section 1.5. The integration of many services in order to provide a simple to use, but expressive application developer interface, resulted in the necessity to implement many framework internal protocols, that govern inter-service cooperation. Splitting most of the services into proxy and worker parts resulted in the need to design and implement service-internal protocols also. Making all the framework internal parts work together and successful hiding of the complexity of framework internals from the application developer is also – from the REMP author's point of view – a significant achievement. The simplicity of framework-provided interfaces helped greatly in implementing the test applications.

The implementation of the prototype was a lengthy process because the code of the services is quite complex. In order to cope with the complexity, the parts of the framework needed to be well-defined and clearly separated. The clear design is also a virtue of the framework.

<u>TA2. Semantics</u>

The second – semantics-related – part of the thesis cannot be proven as directly as the first part. Nonetheless, evaluation use cases, one of which was chosen to be described in

Chapter 5 – showed that the semantic descriptions indeed increase usage flexibility. In case of the presented use case it was not obvious that a five years old PC could serve as the single producer of all messages needed by the consumers. By using concept-based queries, the consumers could be designed so that they did not need to know the number of message producers. Moreover, thanks to the framework's design, their local instances of the REMP Subscribe Service were not aware of the number and placement of the message producers also. That introduced a great level of flexibility at the message producer side.

The introduction of semantics is – in general – a significant advantage of the REMP framework. From the architectural point of view, it could be desirable to integrate the concept-based address resolution with the discovery service. Although – in order not to rely too heavily on a particular middleware platform – the REMP prototype takes a different approach, the integration remains an interesting option for the future.

RG1. Framework architecture design

The architecture of the framework was the main topic of Chapter 3, which discussed issues related to – foreseen – framework instances, and Chapter 4, which presented a prototype built upon an existing technology. It is worth noting, that the architecture of the framework itself was constructed in a way that does not make it dependent from the underlying technologies. Although JXTA provided some services that were reused by the framework prototype, thanks to introducing an abstraction of the reused middleware, implementations based upon other technologies are possible and would not influence end users, i.e. application developers.

RG2. Semantics descriptions and queries

The matching mechanisms designed and implemented inside the framework refer not only to the message descriptions themselves, but also to simple QoS parameters. Moreover, the selection of message sources could be also based upon runtime metrics taken from the framework-provided monitoring and metering service. Therefore, the framework achieved more than that was expected in that area.

RG3. Extensibility and ease of configuration

Most of the framework services are built upon consistently reused, easy to understand, proxy-worker pattern. Therefore, most of the foreseen extensions can be introduced by implementing a worker (or a set of workers). Because framework instance internal entities use dynamic discovery of workers, no changes are needed to the proxy parts of services. To facilitate the development of workers, the prototype provides implementation stubs that deal with the details of protocols that are used by particular services.

For its author, the framework is hard to assess in terms of its ease of configuration. Although the prototype is not equipped with an wizard-like installer, the minimal configuration process seems to be very easy. A REMP-enabled host needs just a few libraries (see Table 3) to be installed and a few environment variables to be set.

Regarding the ease of advanced configuration, each REMP-internal entity is implemented so that it reads a standard *.properties file at its startup. Depending on the entity, various settings are accessible by changing the file. Such approach is widely used in Java applications, so it should be familiar for developers.

RG4. Reuse of existing peer-to-peer technology

The REMP framework reuses JXTA, which claims to be a general-purpose peer-to-peer platform. The framework prototype implementation relies heavily especially on the Resolver and Discovery Services. The former fits the framework perfectly, while the latter needed some extra work mainly because its little support for introducing custom types of advertisements. Moreover, the JXTA's module loading mechanism needed to be enhanced to automatically download generated components from Code Repository Service workers. Despite its deficiencies, in general the platform supports the presented framework greatly.

6.2. Framework shortcomings and future work

This section – based on the identified deficiencies – summarizes the guidelines for framework future development (sections labeled "FD1"-"FD4"). Because the implementation does not cover all features designed by the architecture, most of the suggestions are related to the framework prototype.

FD1. Security

Security is the most underdeveloped aspect of the framework. The JXTA-based prototype allows its user to use only two security-related mechanisms, derived from the underlying middleware: custom group membership service and TLS-based communication channels. In fact, only the latter mechanism is wrapped in the framework implementation as a link property that might be requested by the message consumer issuing a query. Future work in this aspect should also cover the interfaces for the Deployment Service workers administrators to control the privileges of hosted components. Standard JDK mechanisms could be reused for that purpose, but currently are not.

FD2. Deployment and migration

Implementing custom deployment strategies is supported by the prototype, but not to the extent the user would dream of. It would be more convenient for the user if the customization did not need re-compilation of the overlay network manager. Although the operation is not complex, it is the point that requires the user to deal with the framework prototype source code. Given the possibilities offered by contemporary platforms, such strategies could be even loaded dynamically, on user's demand.

Migration of components is defined by the architecture specification, but no automatic migration strategy is supported by the REMP prototype. Moreover, in order to support both deployment and optimization of overlay networks, the overlay network specification could be extended to support deployment and migration strategies selection. Such hints that could govern the selection of a manager for the deployed overlay network.

FD3. Overlay network management service

The current implementation provides the overlay network management service in its simplest form, based on a simple election of an active overlay manager. More sophisticated, load-balancing mechanisms should be provided instead. Moreover, because the managers could use various deployment strategies, the selection of an appropriate manager could be to some extent guided by the aforementioned hints submitted by message consumers. From the framework author's point of view, the area related to the inter-manager cooperation seems to be the most interesting one.

FD4. Support for designing overlay networks

The overlay network specification is a complex document to be filled by each of message consumers. Currently, the only support provided by the framework prototype is based on the XMLBeans-specific API. Although the usage of the interface is intuitive, in order to fill up all the details of the specification the user needs to be focused and patient. GUI-based support for creating overlay networks specifications is very desirable.

References

1. [Arms 1999] W. Arms, *Digital Libraries*, Massachusetts Institute of Technology, 1999
2. [Artymiak 2004] J. Artymiak, *PF + CARP. Firewall 24x7 na platformie OpenBSD. Zastosowania, instalacja, konfiguracja*, meetBSD 2004 Conference, November 27, 2004, Kraków, Poland
3. [Baehni 2004] S. Baehni, P.T. Eugster, R. Gerraoui, *Data-aware multicast*, in Proceedings of the International Conference on Dependable Systems and Networks (DSN 2004)
4. [Baldoni 2003] R. Baldoni, M. Contenti, A. Virgillito, *The Evolution of Publish/Subscribe Communication Systems*, Future Directions of Distributed Computing, Springer Verlag LNCS Vol. 2584, 2003
5. [Baldoni 2004] R. Baldoni, R. Beraldi, L. Querzoni, A. Virgillito, *A Self-Organizing Crash-Resilient Topology Management System for Content-Based Publish/Subscribe*, in A. Carzaniga, P. Fenkam (eds.), Proceedings of the 3rd International Workshop on Distributed Event-Based Systems (DEBS'04), IEEE, 2004
6. [Baldoni 2005] R. Baldoni, A. Virgilito, *Distributed Event Routing in Publish/Subscribe Communication Systems: a Survey*, Technical Report 15-05, Dipartimento di Informatica e Sistemistica, Universita di Roma "La Sapienza", Italy, 2005
7. [Banavar 1999-1] G. Banavar, T. D. Chandra, R. E. Strom, D. C. Sturman, *A Case for Message Oriented Middleware*, Proceedings of the 13th ACM International Symposium on Distributed Computing, Springer-Verlag Lecture Notes in Computer Science, vol. 1693, 1999, p. 1-18
8. [Banavar 1999-2] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, *An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems*, Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, IEEE Computer Society, 1999
9. [Banavar 1999-3] G. Banavar, M. Kaplan, K. Shaw, R. E. Strom, D. C. Sturman, W. Tao, *Information Flow Based Event Distribution Middleware*, Proceedings of 19th IEEE International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-based Applications/Middleware, 1999, p. 114-121
10. [Bartlett 2001] D. Bartlett, *CORBA Junction: CORBA 3.0 Notification Service*, IBM, 2001, http://www-128.ibm.com/developerworks/webservices/library/co-cjct8/
11. [Beckett 2004] D. Beckett, *RDF/XML Syntax Specification*, W3C Recommendation, W3C Consortium, 2004, http://www.w3.org/TR/rdf-syntax-grammar/
12. [Belguidoum 2007] M.Belguidoum and F.Dagnat, *Dependency Management in Software Component Deployment*, Electronic Notes in Theoretical Computer Science vol. 182, Elsevier B.V., 2007
13. [Bhattacharjee 2002] S. Bhattacharjee, *Application-Layer Anycasting*, http://www.matthewjmiller.net/rsrch/497rhk_pres_ppt.pdf
14. [Bhola 2002] S. Bhola, R. E. Strom, S. Baghi, Y. Zhao, J. Auerbach, Exactly-once Delivery in a Content-based Publish-Subscribe System, Proceedings of the International Conference on Dependable Systems and Networks (DSN'2002), IEEE Computer Society, 2002, p. 7-16
15. [Biron 2004] P. Biron (ed.), A. Malhotra (ed.), *XML Schema Part 2 Datatypes Second Edition*, W3C Recommendation, W3C, 2004, http://www.w3.org/TR/xmlschema-2/
16. [Boag 2007] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, XQuery 1.0: An XML Query Language, W3C Recommendation, W3C Consortium, 2007, http://www.w3.org/TR/xquery/
17. [Bobowiec 2006] Ł. Bobowiec, A. Mamuszka, K. Zieliński, S. Zieliński, *Implementation and Performance Study of Reliable Multicast Transport for JXTA*, Proceedings of the Work in Progress Session, 32nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA'06), Cavtat, Croatia, 2006
18. [Bray 2006] T. Bray, D. Hollander, A. Layman, R. Tobin, Namespaces in XML 1.0 (Second Edition), W3C Recommendation, W3C Consortium, 2006, http://www.w3.org/TR/xml-names/

19. [Bray 2008] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, F. Yergeau, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, W3C Recommendation, W3C Consortium, 2008, http://www.w3.org/TR/2008/REC-xml-20081126/
20. [Carzaniga 1998] A. Carzaniga. Architectures for an Event Notification Service Scalable to Wide-area Networks. PhD thesis, Politecnico di Milano, Milano, Italy, 1998.
21. [Carzaniga 2001] A. Carzaniga, D. S. Rosenblum, A. L. Wolf, Design and Evaluation of a Wide-Area Event Notification Service, *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332-383
22. [Carzaniga 2004] A. Carzaniga, M. J. Rutherford, A. L. Wolf, *A Routing Scheme for Content-Based Networking*, Proceedings of IEEE INFOCOM 2004, Hong Kong, China, 2004
23. [Castro 2002] M. Castro, P. Druschel, A.-M. Kermarrec, A. Rowstron, *SCRIBE: A large-scale and decentralized application-level multicast infrastructure*, in IEEE Journal on Selected Areas in Communications, vol. 20, no. 8, October 2002
24. [Castro 2003-1] M. Castro, M. B. Jones, A-M. Kermarrec, A. Rowstron, M. Theimer, H. Wang and A. Wolman, *An Evaluation of Scalable Application-level Multicast Built Using Peer-to-peer overlays*, Proceedings of 22nd Annual Joint Conference of te IEEE Computer and Communications Societies (Infocom), 2003
25. [Castro 2003-2] M. Castro, P. Druschel, A-M. Kermarrec and A. Rowstron, *Scalable Application-level Anycast for Highly Dynamic Groups*, in: B. Stiller, G. Carle, M. Karsten, P. Reichl (Eds.): Group Communications and Charges; Technology and Business Models, 5th COST264 International Workshop on Networked Group Communications, NGC 2003, and 3rd International Workshop on Internet Charging and QoS Technologies, ICQT 2003, Munich, Germany, September 16-19, 2003, Proceedings. Springer 2003, ISBN 3-540-20051-7
26. [Chand 2003] R Chand, P.A. Felber., *A Scalable Protocol for Content-Based Routing in Overlay Networks*, Proceedings of the Second IEEE International Symposium on Network Computing and Applications, IEEE Computer Society, 2003.
27. [Christensen 2004] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Definition Language (WSDL), W3C Consortium, 2004, http://www.w3.org/TR/wsdl
28. [Chu 2002] Y.-H. Chu, S. G. Rao, S. Seshan, H. Zhang, *A case for end system multicast*, IEEE Journal on Selected Areas in Communications, vol. 20, no. 8, pp. 1456–1471, 2002.
29. [Cisco 2005] *Cisco IOS IP Application Services Configuration Guide, Release 12.4*, Cisco Systems, 2005, http://www.cisco.com
30. [Clark 1999] J. Clark, S. DeRose, XML Path Language (XPath) Version 1.0, W3C Recommendation, W3C Consortium, 1999, http://www.w3.org/TR/xpath
31. [Costa 2004] P. Costa, M. Migliavacca, G. P. Picco, G. Cugola, *Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation*, In Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS04), pp. 552-561, IEEE Computer Society Press, 2004
32. [Coulouris 2005] G. Coulouris, J. Dollimore, T. Kindberg, *Distributed Systems, Concepts and Design*, Pearson Education, 2005
33. [Coulson 2002] G. Coulson, *What is Reflective Middleware?*, http://www.comp.lancs.ac.uk/~geoff/Publications/RMARTICLE1.pdf
34. [Credle 2007] R. Credle, J. Adams, K. Clark, Y.P. Ge, H. Jeter, J. Lopes, S. Nasser, K. Peri, Patterns: SOA Design Using WebSphere Message Broker and WebSphere ESB, IBM, 2007, http://www.redbooks.ibm.com/redbooks/pdfs/sg247369.pdf
35. [Cugola 2001] G. Cugola, E. Di Nitto, A. Fuggetta, *The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS*, IEEE Transactions on Software Engineering, September 2001, Vol. 27, No. 9, pp. 827-850
36. [Cugola 2002] G. Cugola, H.-A. Jacobsen, *Using publish/subscribe middleware for mobile systems*, ACM SIGMOBILE Mobile Computing and Communications Review, Vol. 6, Issue 4, 2002
37. [Davies 2005] S. Davies, P. Broadhurst, WebSphere MQ v6 Fundamentals, IBM 2005, www.redbooks.ibm.com

38. [Deering 1990] S. E. Deering, D. R. Cheriton, Multicast routing in datagram networks and extended LANs, *ACM Transactions on Computer Systems*, vol. 8, no. 2, 1990, pp. 85-111

39. [Deng 2001] J. Deng, *Jena – A Java API for RDF*, University of Colorado at Boulder, 2001, www.cs.colorado.edu/~kena/classes/7818/f01/presentations/jena.ppt

40. [Denning 2006] Infoglut, Communications of the ACM, July 2006, vol. 49, no. 7

41. [Diao 2004] Y. Diao, S. Rizvi, M.J. Franklin, *Towards an Internet-Scale XML Dissemination Service*, Proceedings of the Thirtieth international conference on Very large data bases, VLDB Endowment, Inc., 2004.

42. [DOM 1998] A. Le Hors et. al. (ed.), *Document Object Model (DOM) Level 3 Specification*, W3C Recommendation, W3C, 1998

43. [Eisenbach 2002] S. Eisenbach, C. Sadler, S. Shaikh, *Evolution of Distributed Java Programs*, in J. Bishop (ed.), Component Deployment 2002, Lecture Notes in Computer Science, Springer-Verlag, Germany, 2002

44. [Elvin-sub] Elvin Subscription Language Reference, at Avis event router project website, http://avis.sourceforge.net/subscription_language.html

45. [Fikes 2003] R. Fikes, P. Hayes, I. Horrocks, *OWL-QL - A Language for Deductive Query Answering on the Semantic Web*, Knowledge Systems Laboratory, Stanford University, Stanford, USA, 2003.

46. [Fischer 2005] F. Fischer, Applied Computer Science Problems: DCOM, Leopold-Franzens Universität Innsbruck, 2005, http://www.sti-innsbruck.at/fileadmin/documents/teaching_archive/acsp0405/11_Fischer_Ausarbeitung.pdf

47. [Flenner 2003] R. Flenner et al., *Java™ P2P Unleashed*, Sams Publishing, USA, 2003, ISBN: 0-672-32399-0

48. [FUSE 2009] Exploring JMS with FUSE™ Message Broker, Version 5.3, Progress Software, 2009, http://fusesource.com/docs/broker/5.3/exploring_jms/explore_jms.pdf

49. [Gore 2001] P. Gore, R. Cytron, D. Schmidt, C. O'Ryan, *Designing and Optimizing a Scalable CORBA Notification Service*, Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems, USA, 2001, p. 223-229

50. [Gray 2004] N. A. B. Gray, Comparison of Web Services, Java-RMI, and CORBA service implementations, Fifth Australian Workshop on Software and System Architectures, Melbourne, Australia, 2004, http://mercury.it.swin.edu.au/ctg/AWSA04/Papers/gray.pdf

51. [Haase 2004] P. Haase, J. Broekstra, A. Eberhart, R. Volz, *A comparison of RDF query languages*, Proceedings of the Third International Semantic Web Conference, Hiroshima, Japan, November 2004

52. [Haase 2005] P. Haase, *A Comparison of RDF Query Languages*, 2005, http://www.aifb.uni-karlsruhe.de/WBS/pha/rdf-query/

53. [Hapner 2002] M. Hapner, R. Burridge, R. Sharma, J. Fialli, K. Stout, Java Message Service Specification, version 1.1, Sun Microsystems, 2002, http://java.sun.com/products/jms/

54. [Hashimoto 2006] M. Hashimoto, S. Ata, H. Kitamura, M. Murata, *IPv6 Anycast Terminology Definition*, IETF Internet Draft, 2006, http://www.ietf.org/internet-drafts/draft-doi-ipv6-anycast-func-term-05.txt

55. [Hinden 2004] R. Hinden (ed.), *Virtual Router Redundancy Protocol*, RFC 3768, IETF, 2004

56. [Hinden 2006] R. Hinden, S. Deering, IP Version 6 Addressing Architecture, RFC 4291, IETF, 2006

57. [Hohpe 2003] G. Hohpe, B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley, 2003, ISBN 0321200683

58. [IBM 2006] *Integrated Ontology Development Toolkit documentation*, Version 1.1.2, IBM, 2006

59. [IONA 2000] IONA Technologies PLC., *Orbix Programmer's Guide, Java Edition*, IONA 2000, http://www.iona.com/support/docs/orbix/gen3/33/pdf/orbix33java_pguide.pdf

60. [Janotti 2000] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, J. W. O'Toole, *Overcast: reliable multicasting with an overlay network*, in: Proceedings of the 4th Symposium on Operating System Design and Implementation (OSDI '00), pp. 197–212, San Diego, CA, USA, October 2000.
61. [Jin 2006] X. Jin, K.-L. Cheng, and S.-H. Gary Chan, *SIM: scalable island multicast for peer-to-peer media streaming*, in Proceedings of IEEE International Conference on Multimedia & Expo (ICME '06), pp. 913–916, Toronto, Canada, July 2006.
62. [Jin 2007] X. Jin, K.-L. Cheng, S.-H. Gary Chan, *Scalable Island Multicast for Peer-to-Peer Streaming*, in: Advances in Multimedia, Hindawi Publishing Corporation, 2007
63. [JXTA-met 2003] https://jxse-metering.dev.java.net/docs/MeteringOverview.pdf
64. [JXTA 2004] *JXTA v2.0 Protocols Specification*, The Internet Society, 2004, https://jxta.dev.java.net/
65. [JXTA 2007] *JXTA Java Standard Edition v2.5: Programmers Guide*, Sun Microsystems, 2007
66. [JXTA-RM 2005] The JXTA-RM Project website, http://jxta-rm.jxta.org
67. [Keeney 2008] J. Keeney, D. Roblek, D. Jones, D. Lewis, D. O'Sullivan, *Extending Siena to support more expressive and flexible subscriptions*, Proceedings of the second international conference on Distributed event-based systems, ACM, Rome, Italy, 2008, ISBN:978-1-60558-090-6
68. [Klyne 2004] G. Klyne (ed.), J. Carroll (ed.), *Resource Description Framework (RDF): Concept and Abstract Syntax*, W3C Recommendation, W3C, 2004
69. [Kon 2002] F. Kon, F. Costa, G. Blair, R. H. Campbell, *The Case for Reflective Middleware*, Communications of the ACM, 45(6), p. 33-38, USA, 2002
70. [Kreger 2003] H. Kreger, W. Harold, L. Williamson, *Java™ and JMX. Building Manageable Systems*, Addison-Wesley, 2003
71. [Leach 2005] P. Leach, M. Mealling, R. Salz, *A Universally Unique IDentifier (UUID) URN Namespace*, IETF, 2005
72. [LeSommer 2006] N. Le Sommer, F. Guidec, H. Roussain, *A context-aware middleware platform for autonomous application services in dynamic wireless networks*, Proceedings of the first international conference on Integrated internet ad hoc and sensor networks, ACM, 2006
73. [Li 1998] T. Li, B. Cole, P. Morton, D. Li, *Cisco Hot Standby Router Protocol*, RFC 2281, IETF, 1998
74. [Loyall 2001] J. Loyall et al., *Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications*, in Proceedings of the 21st International Conference on Distributed Computing Systems, IEEE Computer Society, USA, 2001
75. [Ławniczek 2003] B. Ławniczek, G. Majka, P. Słowikowski, S. Zieliński, K. Zieliński, *Grid Infrastructure Monitoring Service Framework - Jiro/JMX Based Implementation*, Electronic Notes in Theoretical Computer Science, vol. 82, issue 6, Elsevier B. V., 2003
76. [Mahmoud 2004] Q. S. Mahmoud (ed.), Middleware for Communications, John Wiley & Sons, 2004
77. [McGuinness 2004] D.L. McGuinness, F. van Harmelen (eds.), *OWL Web Ontology Language Overview*, W3C Recommendation, W3C Consortium, 2004, http://www.w3.org/TR/owl-features/
78. [McLaughlin 2006] B. McLaughlin, J. Edelson, Java and XML, O'Reilly Media, Inc., 2006
79. [MCM 2008] 9th Milano City Marathon Timing Data Service website, http://www.tds-live.com/wtrpg/race.jsp?id=2175
80. [Milojicic 2002] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, Z. Xu, *Peer-to-Peer Computing*, HP Laboratories Palo Alto, 2002
81. [MIX] MIX – A Data Model for Semantic Data Integration, Technische Universität Darmstadt, http://www.dvs.tu-darmstadt.de/research/mix/
82. [Moats 1997] R. Moats, *URN Syntax*, IETF, 1997
83. [Mühl 2002] G. Mühl, *Large-Scale Content-Based Publish/Subscribe Systems*, PhD Dissertation, Technische Universität Darmstadt, 2002

84. [Narten 2007] T. Narten, E. Nordmark, W. Simpson, H. Soliman, *Neighbor Discovery for IP version 6 (IPv6)*, RFC 4861, IETF, 2007

85. [ObjC 2008] *The Objective-C 2.0 Programming Language*, Apple Inc., 2008, http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf

86. [OMG 2000-1] CORBA Event Service Specification, Version 1.1, Object Management Group, 2000

87. [OMG 2000-2] CORBA Trading Object Service Specification, version 1.0, OMG, 2000

88. [OMG 2004] The Common Object Request Broker Architecture: Core Specification, Revision 3.0.3, Object Management Group, 2004

89. [OMG 2004] CORBA: Notification Service, Version 1.1. Specification, Object Management Group, 2004

90. [OMG-UML-S 2007] *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*, Object Management Group (OMG), 2007, http://www.omg.org/docs/formal/07-11-01.pdf

91. [OpenBSD 2003] *OpenBSD Programmer's Manual*, Berkeley Software Design, Wolfram Schneider, 2003, http://www.openbsd.org/cgi-bin/man.cgi

92. [Ort 2003] E. Ort, B. Mehta, Java Architecture for XML Binding, Sun Microsystems, 2003, http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

93. [Partridge 1993] C. Partridge, T. Mendez, W. Milliken, *Host Anycasting Service*, RFC 1546, IETF, 1993

94. [Patel-Schneider 2004] P.F. Patel-Schneider, P. Hayes, I. Horrocks (eds.), *OWL Web Ontology Language Semantics and Abstract Syntax*, W3C Recommendation, W3C Consortium, 2004, http://www.w3.org/TR/owl-semantics/

95. [Perry 2002] J. S. Perry, Java Management Extensions, O'Reilly Media, Inc. 2002, ISBN: 978-0-596-00245-9

96. [Pierce 2002] B. C. Pierce, *Types and Programming Languages*, MIT Press, 2002, ISBN 0-262-16209-1

97. [Pietzuch 2004] P. R. Pietzuch, Hermes: A scalable event-based middleware, Technical Report UCAM-CL-TR-590, University of Cambridge, 2004

98. [Pissias 2008] P. Pissias, G. Coulson, A Framework for Quiescence Management in Support of Reconfigurable Multithreaded Component-based Systems, IET Software, Vol 2, No 4, 2008, p. 348-361

99. [Plummer 1982] D. C. Plummer, *An Ethernet Address Resolution Protocol*, RFC 826, IETF, 1982

100. [Powers 2003] S. Powers, *Practical RDF*, O'Reilly & Associates, Inc., 2003, ISBN 0-596-00263-7

101. [Prud'hommeaux 2008] E. Prud'hommeaux, A. Seaborne, SPARQL Query Language for RDF, W3C Recommendation, W3C Consortium, 2008, http://www.w3.org/TR/rdf-sparql-query/

102. [RacerPro 2007] RacerPro User's Guide, Version 1.9.2, Racer Systems GmbH & Co. KG, 2007, http://www.racer-systems.com

103. [RAP] RAP – RDF API for PHP, http://www.seasr.org/wp-content/plugins/meandre/rdfapi-php/doc/

104. [Raspet 2004] S. Raspet, CARP your way to high availability, Sourceforge Inc., http://www.linux.com/feature/35482

105. [Redkar 2004] A. Redkar, C. Walzer, S. Boyd, R. Costall, K. Rabold, T. Redkar, *Pro MSMQ: Microsoft Message Queue Programming*, Apress, 2004, ISBN 1590593464

106. [Saraswat 1997] V. Saraswat, *Java is not type-safe*, AT&T Research, 1997, http://www.cis.upenn.edu/~bcpierce/courses/629/papers/Saraswat-javabug.html

107. [SAS 2008] *SAS Integration Technologies: Developer's Guide*, SAS Institute, http://support.sas.com/rnd/itech/doc9/dev_guide/

108. [Segall 2000] B. Segall, D. Arnold, J. Boot, M. Henderson, T. Phelps, *Content Based Routing with Elvin4*, Proceegings of the Australian Unix and Open Systems Users Group (AUUG) Annual Conference, Australian National University, Canberra, Australia, 2000

109. [SETI] SETI@home project website, http://setiathome.ssl.berkeley.edu/

110. [Shirky 2000] C. Shirky, *What is P2P…And What Isn't*, An article published on O'Reilly Network, http://www.openp2p.com/pub/a/472

111. [Sirin 2007] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, Y. Katz, *Pellet: A practical OWL-DL reasoner*, Web Semantics: Science, Services and Agents on the World Wide Web, Vol. 5 Issue 2, 2007, p. 51-53

112. [Srivatsa 2005] M. Srivatsa, L. Liu, *Securing publish-subscribe overlay services with EventGuard*, Proceedings of the 12th ACM conference on Computer and communications security, USA, 2005

113. [Stoica 2003] Ion Stoica, *Overlay Networks*, in J. Liebeherr (ed.) Computer Networks lecture notes, University of Virginia, 2003, http://www.cs.virginia.edu/~cs757/slidespdf/757-09-overlay.pdf

114. [Sturman 1998] D. Sturman, G. Banavar, R. Strom, *Reflection in the Gryphon Message Brokering System*, Proceedings of the Reflection Workshop of 13th ACM Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA), ACM, 1998

115. [Sun 1999] Java Remote Method Invocation (RMI). Specification, Sun Micosystems, 1999

116. [Tannenbaum 2002] A. S. Tannenbaum, M. V. Steen, Distributed Systems: Principles and Paradigms, Prentice Hall, 2002

117. [Terry 2002] S. Terry, Enterprise JMS Programming, M&T Books, 2002, ISBN: 0-7645-4897-2

118. [Thompson 2004] H. S. Thompson (ed.), D. Beech (ed.), M. Maloney (ed.), N. Mendelsohn (ed.), *XML Schema Part 1: Structures Second Edition*, W3C Recommendation, W3C, 2004, http://www.w3.org/TR/xmlschema-1/

119. [TIBCO 2007] TIBCO Rendezvous, http://www.tibco.com/software/messaging/rendezvous/

120. [Verzulli 2001] J. Verzulli, *Using the Jena API to Process RDF*, O'Reilly Media Inc., 2001, http://www.xml.com/pub/a/2001/05/23/jena.html

121. [Waldo 2003] J. Waldo, The Jini™ Specifications, 2nd ed., Sun Microsystems, 2003, ISBN: 0-201-72617-3

122. [Zhao 2001] Y. Zhao, R. E. Strom, *Exploiting Event Stream Interpretation in Publish-Subscribe Systems*, Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC 2001), ACM, 2001, p. 219-228

123. [Zhou 2006] Y. Zhou, B. C. Ooi, K.-L. Tan, F. Yu, *Adaptive reorganization of coherency-preserving dissemination tree for streaming data*, in L. Liu, A. Reuter, K.-Y. Whang, J. Zhang (eds.), Proceedings of the 22nd International Conference on Data Engineering (ICDE'06), IEEE Computer Society, 2006