Report

A Pattern Language for Overlay Networks Design Patterns in Peer-to-Peer Systems

Author(s):
Grolimund, Dominik

Publication Date:
2005

Permanent Link:

https://doi.org/10.3929/ethz-a-006788249 →

Design Patterns in Peer-to-Peer Systems

A Pattern Language for Overlay Networks

Technical Report 503
Department of Computer Science
ETH Zurich

Dominik Grolimund
dominik.grolimund@inf.ethz.ch

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Developing peer-to-peer systems is a big challenge, because they typically operate in large-scale, highly unreliable and insecure environments and involve the collaboration of many nodes (i.e. computers) in the network. In recent times, a lot of efforts have been devoted to the system design of peer-to-peer systems, resulting in new structured overlay networks. These overlay networks perform the routing task needed to localize items in the network efficiently even in the presence of node failures and attacks. However, to tackle the full complexity of practical peer-to-peer systems, also good software design needs to be applied. Yet, lots of peer-to-peer systems are developed in an ad-hoc manner, and little has been written about their software architecture. In this paper, I have tried to analyze the software design of overlay networks, the common abstraction of most peer-to-peer systems. Design patterns have been very successful in documenting proven and reusable solutions; consequently, I have tried to discover design patterns for overlay networks by investigating different academic and open source projects, and applying and improving them on a real-world system. This paper presents the result of this work: a pattern language for overlay networks, consisting of a number of patterns and their relationships, trying to cover most aspects of overlay networks. For that reason, most patterns are also not new, but are simply adaptations of well-known patterns to the specific requirements of overlay networks. The remaining patterns can be regarded as suggested proto-patterns.

Contents

1 Introduction **3**

2 Overview **6**
2.1 Peer-to-Peer Systems 6
2.2 Overlay Networks 7
2.3 Patterns 7
2.4 Pattern Languages 8

3 Related Work **9**

4 Method **12**

5 Design Issues **14**

6 Pattern Language **19**
6.1 Pattern Form 19
6.2 Overview 21
6.3 Summaries 23

7 Patterns **28**
7.1 Application Interaction 28
7.1.1 Overlay Facade 28
7.1.2 Application Delivery 32
7.1.3 Application Notification 36
7.1.4 Extended Overlay Facade 40
7.1.5 Abstract Address Handle 44
7.2 Messages 48
7.2.1 Message Factory 48
7.2.2 Envelope Wrapper 52
7.2.3 Basic Message 54
7.2.4 Control Message 57
7.2.5 Routed Message 59

7.2.6 Specialized Message Type . . . 62
7.2.7 Source Sink Marker . . . 67
7.3 Message Handling . . . 71
7.3.1 Message Dispatcher . . . 72
7.3.2 Message Handler . . . 76
7.3.3 Autonomous Message . . . 79
7.3.4 Message Verifier . . . 84
7.4 Routing . . . 90
7.4.1 Router . . . 90
7.5 Local Node . . . 98
7.5.1 Local Node . . . 98
7.5.2 Local Node For Each Type . . . 101
7.6 Protocol . . . 104
7.6.1 Self Maintenance . . . 105
7.6.2 Separate Protocol . . . 110
7.7 Remote Nodes . . . 113
7.7.1 Node Handle . . . 113
7.7.2 Typed Node Handle . . . 116
7.7.3 Node Handle Proxy . . . 119
7.8 Network Interaction . . . 122
7.8.1 Network Gateway . . . 122
7.8.2 Network Stub . . . 126
7.8.3 Traffic Monitor . . . 128

8 Example **132**

9 Conclusions **134**

Bibliography **136**

1 Introduction

In recent times, peer-to-peer systems have gained a lot of popularity due to file sharing applications such as Napster, Gnutella, eDonkey, Kazaa, and others. Basically, peer-to-peer refers to a decentralized architecture in which all nodes (i.e. computers) have identical capabilities and responsibilities and all communication is potentially symmetric. Peer-to-peer systems have the unique advantage of being able to harness idle resources (computation cycles, bandwidth, and storage) of participating computers at the edge of the Internet. This implies that they typically work in a large-scale, highly unreliable and insecure environment.

Developing such a system is a big challenge. While early peer-to-peer applications have been developed rather experimentally, they have attracted a great deal of attention from computer science research ever since, resulting in a number of ongoing projects at leading universities around the world. Most of this research activity has been focused on the routing problem: Given a key, find the node that is responsible for that key. This has led to new structured overlay networks (e.g. Chord [1], Pastry [2], Tapestry [3], Kademlia [4], CAN [5]), which solve this task efficiently. Other research projects are dealing with security and stability issues in the presence of attacks and node failures.

Altogether, almost all research efforts have been devoted to system design so far. However, a system exposed to such an environment also needs to be well designed from a software engineering point of view. Yet, lots of existing peer-to-peer applications are developed in an ad-hoc manner, and little has been written about their software architecture. Part of this is probably because good design solutions for traditional distributed systems (client/server) have been around for a long time (e.g. [6]), and lots of them can be adapted for peer-to-peer systems as well. On the other hand, peer-to-peer systems exhibit a number of characteristics that are very different from centralized, asymmetric distributed systems, which must have a manifestation in the software design as well. This observation is also supported by peer-to-peer framework initiatives, such as Sun's JXTA [7], that are building a higher abstraction around the core problems of peer-to-peer systems.

Although frameworks can be applied to implement a variety of different systems, it is sometimes questioned whether they are suitable for highly specialized

applications [8]. For that reason, large-scale overlay networks are often developed from scratch. What is needed to improve the software design of peer-to-peer systems in general, are smaller-scale architectural elements which represent good solutions in specific contexts. This is the abstraction level of design patterns, which have proven very successful in documenting solutions in a number of domains.

In this paper, I am trying to find proven design solutions for recurring problems in peer-to-peer systems. Consequently, I have tried to identify design patterns. Because patterns are rooted in practice, I have investigated different academic and open-source projects, as well as tried to implement and improve found design patterns on a real-world project. This paper presents the results of this work: a pattern language for peer-to-peer systems, describing a number of patterns that reoccur in different projects and have led to a favorable design. It focuses on the common abstraction of most peer-to-peer systems, the overlay network layer, which is responsible for implementing the routing algorithm. Because it is trying to solve design issues for most aspects of overlay networks, most patterns are also not new, but either represent a standard object-oriented design solution, a well-known, existing pattern, or an adpatation of a well-known pattern to the specific needs of overlay networks. The remaining patterns can be regarded as suggested proto-patterns (potential pattern candidates, see for instance [9]), which are a generalization of ideas that have led to a favorable design in some overlay networks.

The contributions of this paper and the pattern language are therefore twofold: First, it tries to capture the expertise and best practice of software designers and peer-to-peer programmers, and collects this knowledge in a pattern language for most aspects of overlay networks. Second, it presents a number of suggested proto-patterns, which have not yet been documented, but belong to the fundemantal building blocks of overlay networks.

The primary target audience of this paper is programmers and software designers of peer-to-peer systems. The experienced programmer of peer-to-peer applications will probably find that most patterns are familiar to him. For him, the values of a pattern language are that these design solutions are generalized and collected, so that they can help communicate design ideas to others. For someone new to peer-to-peer systems, it should be a helpful guide when designing the application. The paper might also be interesting to software designers and programmers from other domains, who are curious to have a look at the software architecture of peer-to-peer applications. Finally, I also hope to find pattern enthusiasts reading this paper and giving me critical feedback on the patterns.

The rest of this paper is structured as follows. Section 2 gives a brief overview over peer-to-peer systems, overlay networks, patterns and pattern languages. It can be freely skipped by a reader who is familiar with these topics. In section 3, the related work by others is outlined, and in section 4, I describe the method I

have used to find the patterns. Section 5 analyzes the design issues involved in the software design of an overlay network. The main part of this paper is given in sections 6 and 7, which explain the pattern language and the patterns for overlay networks. Section 6 gives an overview over the pattern language, and section 7 describes each pattern in detail. In section 8, I show how the different patterns can be combined together to a skeleton of an actual overlay network. Finally, in section 9, I conclude by analyzing briefly which design issues have been solved by the pattern language, and the future work that remains to be done.

2 Overview

This section gives an overview over the different topics that make up this work. However, they are not treated in detail, and the familiar reader can freely skip them. Section 2.1 gives a short introduction into peer-to-peer systems, and section 2.2 goes on to describe the common core of most peer-to-peer systems, the overlay network. In section 2.3, I give a short overview and definition of patterns. Finally, section 2.4 explains how collections of patterns are combined in a pattern language.

2.1 Peer-to-Peer Systems

Clay Shirky defines peer-to-peer as follows [10]. 'Peer-to-peer is a class of applications that takes advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, peer-to-peer nodes must operate outside the DNS system and have significant or total autonomy from central servers.'

This definition illustrates peer-to-peer systems as an application-level Internet on top of the Internet. From a bird's eye view, a peer-to-peer system simply consists of lots of nodes that are communicating together, both as clients and servers, in terms of requesting and sending data. Of course, this concept is not new, and big parts of the Internet infrastructure itself are peer-to-peer as well. What is new is the fact that the nodes in the system are at the edge of the Internet and thus unreliable personal computers, which can be turned on and off at any time. Peer-to-peer systems therefore need to cope with these inherent dynamics. It also implies that they are very exposed to attacks and failures, because they operate in an open and insecure environment. Moreover, the scale of peer-to-peer systems can be extremely large, incorporating millions of nodes.

Peer-to-peer applications have been built for different purposes. An obvious application is file sharing, which has stirred quite a lot of controversy in the last years. Other applications enable communication and collaboration (e.g. Skype [11], Jabber [12]) in a peer-to-peer manner. Although their purpose is different, they all share the need to localize items in the network (e.g. fragments, files,

users). Napster, one of the first file sharing applications, used a simple approach; localization was not decentralized, but provided by a central server. Unfortunately, such a simple approach is not scalable. Early systems that followed Napster, such as Gnutella [13], tried to overcome this scalability problem by using a decentralized approach; a query was simply flooded through the system. A single query therefore resulted in lots of messages and was also not efficient in terms of the necessary search steps [14]. To improve the routing efficiency, computer science research has come up with structured overlay networks, which are explained in the next section. For a deeper introduction into peer-to-peer systems, see for instance [15], [16], [17].

2.2 Overlay Networks

Structured overlay networks, such as Chord [1], Pastry [2], Tapestry [3], Kademlia [4], or CAN [5], are responsible for implementing an efficient routing algorithm. The nodes in the system are structured in order to decrease the search steps necessary to find the target identifier. Each node maintains a local routing table, which holds the identifiers of other nodes in the system. When a query message arrives, the node forwards the message to the one from its local routing table, which is closest (using an appropriate metric) to the specified target identifier. The routing complexity is typically O(log n), where n is the number of nodes in the system, while each node is maintaining O(log n) nodes in its routing table.

2.3 Patterns

Patterns are a way of 'documenting experience by capturing successful solutions to recurring problems' [18]. Therefore, they are best suited for the goal of this work, describing proven solutions for design problems in peer-to-peer systems.

Although patterns are well-known in software engineering, especially due to the success of the book 'Design Patterns' by the 'Gang Of Four' [19], they have successfully been applied to other domains as well, including patterns for organizations, processes, pedagogics, analysis, customer interaction, and many more [20]. In fact, the concept of patterns found its way into software engineering originally from architecture, when Christopher Alexander was studying ways to improve the process of designing buildings and urban areas, describing solutions as a set of patterns.

In his books ([21], [22]), Christopher Alexander defines patterns as follows. 'Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the

same way twice'. The Hillside Group [20], which organizes pattern workshops for pattern writers around the world, gives the following definition. 'Each pattern is a three-part rule, which expresses a relation between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain software configuration which allows these forces to resolve themselves'.

2.4 Pattern Languages

Patterns rarely stand in isolation, but come in collections with rules to combine them. Christopher Alexander coined the term 'pattern language' to refer to such a collection of patterns. Unlike a mere pattern compilation or catalog, a pattern language includes rules and guidelines which explain how and when to apply its patterns to solve a problem which is larger than any individual pattern can solve [23]. Pattern languages organize the expertise of a domain and create a common vocabulary, which helps communicate software design.

3 Related Work

So far, little has been written about the software design of peer-to-peer systems. To the best of my knowledge, no patterns for peer-to-peer systems have been documented yet. The goal of this work is to identify patterns for overlay networks, the common abstraction of most peer-to-peer systems. In this section, however, I want to outline the related work in a broader perspective.

The closest related work I could find was the investigation of patterns in peer-to-peer systems of a focus group at EuroPlop 2002 [24], the European patterns workshop organized by the Hillside Group. A focus group is a discussion session, in which problems and possible patterns of a domain are discussed. In their paper [25], which describes the result of the discussion brain storming, they outline different characteristics of peer-to-peer systems and possible problems from a very broad perspective. They concluded that new patterns might be discovered, but that most issues could be solved with well-known patterns from distributed systems. It never resulted in an actual work on that topic or in a documentation of patterns so far.

When writing this paper, I found that EuroPlop 2005 was taking up on this. They organized another focus group on the topic of peer-to-peer systems. So far, the results of this group session have not been published. However, from their goal description [26], I assume that the scope is still much broader than I take in here, where I focus only on overlay networks. Therefore, I am very interested in seeing those results and hope that they can complement the patterns presented in here to cover a wider area of problems in peer-to-peer systems.

In February 2005, after the task description of this paper was online, I received an e-mail from a German student who was writing his Master thesis on a very similar topic. Unfortunately, I have not yet received his thesis paper and could not find it on the Internet either.

Related to the architecture of peer-to-peer systems is also the effort by different research groups to agree on a common application programming interface (API) for overlay networks. In their paper [27], they also indicate that overlay networks are the key component of most peer-to-peer systems and show how different applications can be built on top, including examples such as distributed hash tables,

decentralized object location and routing, and group anycast / multicast. On top of these higher level abstractions, actual applications such as CFS [28], PAST [29], Scribe [30], and OceanStore [31] can be developed. In their paper, the overlay network as defined in here is referred to as the key-based routing layer.

In [32], a socket-based approach for programming applications build on top of overlay networks has been proposed, which has been successfully used for implementing HyperCast [33]. The overlay socket API describes how applications can interact with different overlay network protocols, and how different transport protocols can be used. Compared to this work, the focus is more on this interaction, rather than on the design of the overlay network itself.

The biggest effort in building a higher abstraction around the core problems of peer-to-peer systems is probably project JXTA [7], a set of open protocols that allow any device to communicate and collaborate in a peer-to-peer manner. JXTA is a specification of a peer-to-peer infrastructure layer on top of the network layer, and is therefore closely related to the notion of an overlay network. However, the scope and tasks of this layer in JXTA are much broader than typically found in overlay networks, including entities such as groups, advertisements, services, and the like.

To build a fully functional peer-to-peer system, one does not only need to implement the overlay network of course. Therefore, other problems will appear throughout the whole design. However, good design solutions for problems at the application level are well-known, and are probably not specific to peer-to-peer applications. On the other hand, a rich set of patterns from distributed systems in general are available for problems arising at the network layer. I want to conclude this section with a (incomprehensive) list of such problems together with some references on where solutions can be found.

- **Concurrency:** Concurrency is an inherent issue in any distributed system. In peer-to-peer systems, many blocking operations are involved, so that they benefit from concurrency even on a single processor machine. This topic has been researched in detail. 'Design Principles for Concurrent Systems' by Doug Lea [34] describes a number of best practices and design patterns for concurrent systems. Additionally, a lot of the patterns in 'Pattern-Oriented Software Architecture II' [6] provide solutions for designing scalable concurrent distributed systems.

- **Messaging:** Nodes in peer-to-peer systems communicate by exchanging messages. Consequently, they face similar problems as all applications based on messages. Messaging, as it is often referred to, has been investigated extensively in the light of enterprise application integration. Several books exist on that topic. One with a special treatment on design issues and solu-

tions, taking a pattern approach, is the recently published book 'Enterprise Integration Patterns' [35].

- **Asynchronous operations:** Sending messages is inherently asynchronous, which results in a number of design problems when offering these services to an application. Again, this issue is present in all distributed systems, so that this topic has been researched in detail as well. In overlay networks, asynchronous operations turned out to be no design problem, because it still operates at the level of messages. At the application level, or the thin layer of distributed hash tables, these issues become more relevant, because they build a higher abstraction further away from the notion of messages. Good design patterns for asynchronous systems can be found in 'Pattern-Oriented Software Architecture II' [6].

- **Security:** This is an issue present in all distributed systems as well, because they are more open than other software systems. Peer-to-peer systems are especially exposed. However, most of these issues are rather a matter of system design, not software design.

- **General Design Patterns:** At the application level, but also throughout the software design, general design problems will arise which are not specific to the domain. The most famous book on general design patterns is 'Design Patterns' by the 'Gang of Four' [19], and lots of tutorials and learning guides exist on the patterns presented in that book. To name just two websites on design patterns: the 'Portland Pattern Repository' [36] by Ward Cunningham provides a huge resource, and the website of the HillSide Group [20] is a good starting point to explore the vast area of patterns.

- **Application Architecture:** When building a large application, design problems will show up at the architectural level. Even though it focuses on enterprise applications, the book 'Patterns for Enterprise Application Architectures' by Martin Fowler [37] is an invaluable reference for any large, structured and layered software architecture.

4 Method

The term 'pattern mining' is often used to describe the process of looking for patterns to document [18], [38]. It typically involves examining lots of source code and distilling out the core elements that have led to a good design. This indicates that patterns are rooted in practice, which was also the way I came to do this work.

In autumn 2004, I started work on a distributed hash table at ETH Zurich [39], which includes a new overlay network as well. After the system design and simulations have been done, the software had to be designed and implemented. I reasoned about the core problems that would occur in the design and did a literature research, trying to find design solutions for peer-to-peer systems. Unfortunately, I could not find a lot on this topic. However, the closely related field of distributed systems has a rich set of design patterns (e.g. [6]), and some of them could be applied in this system as well. However, yet other issues at the conceptual level of the overlay network remained unsolved by known patterns.

After implementing the overlay network, I knew the core problems and was curious to see how others solved them. I was very motivated to do this investigation in more detail, trying to find generalized design solutions that could help others with the same problem. I organized the source code of different open source and academic peer-to-peer projects. Lots of them turned out to be implemented in an ad-hoc manner, so that no general design solutions could be found. Others, however, were trying to model the overlay network to solve the core problems in the best way. Finally, I investigated a number of projects written in Java, including FreePastry [40], Tapestry [41], Bamboo [42] (based on Tapestry), P-Grid [43], a Viceroy implementation [44], JXTA [7], LimeWire [45], jMule [46], Dijjer [47], OogP2P [48], GISP [49], Azureus [50], JTorrent [51], and Meteor [52]. Other interesting projects that I knew about were either not open source, or written in C (e.g. Chord [53]), which did not yield much hope of good design solutions at the object level.

Because of my experience in developing a distributed hash table (including the underlying overlay network), I focused more on very similar projects, rather than other projects, for example about communication or grid computation. Therefore,

the projects I have investigated are more data-centric than for instance a chat application, which needs to localize users instead of data fragments. It is likely that different problems appear in these applications, even though I believe that the overlay network will still remain quite similar. This belief is also based on the fact that Pastry [2], a structured overlay network, has been successfully used for different applications, including PAST [29], a distributed archival storage system, Scribe [30], a group communication system, and at least four other applications as well [54]. Another strong indication that this assumption is correct are the efforts made by different research groups to agree on a common application programming interface (API) for overlay networks [27].

Whenever I found the slightest hint of a pattern, I iterated back and forth between examining that source code and implementing it, so that I could immediately see if the solution was beneficial in a different context as well.

Of all projects, the most efforts have probably been put into project JXTA, the peer-to-peer framework initiated by Sun. As a framework, it needs to be flexible to allow any kind of application to be built on top of it. Because of this generic approach, JXTA's architecture is much more complex than the highly specific overlay networks developed by other research groups. It deals with entities such as advertisements, groups, and services, which are not present in other projects. For this reason, I have not investigated JXTA in detail. However, it would be very interesting to 'mine' JXTA for design patterns for a wider set of problems of peer-to-peer systems.

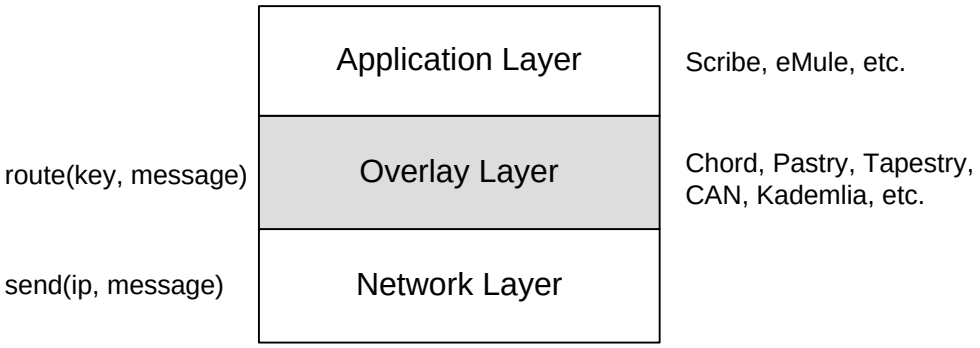
Real world projects face some additional problems that lots of research projects do not take into account. Even though in theory peer-to-peer systems are totally symmetric, in reality one will soon need to define different roles for the nodes. This is because the different computers connected to the network are completely heterogeneous, so that it is not possible to treat them identical. One example is problems with firewalls and network address translators (NAT), which make it sometimes impossible to send unsolicited messages to a node. Therefore, such computers cannot participate in the routing process. However, this is not the only reason to define different roles. Another example are computers with much more resources at their disposal (e.g. bandwidth, storage), so that they can be assigned with additional tasks to improve overall efficiency. Whereas these problems are rarely taken into account in research projects, they are often solved in open source projects, so that I had a closer look at them as well.

Altogether, I only present patterns which I also tried to apply, so that I properly understood the design problem, the trade-offs (often called forces), and the solution.

5 Design Issues

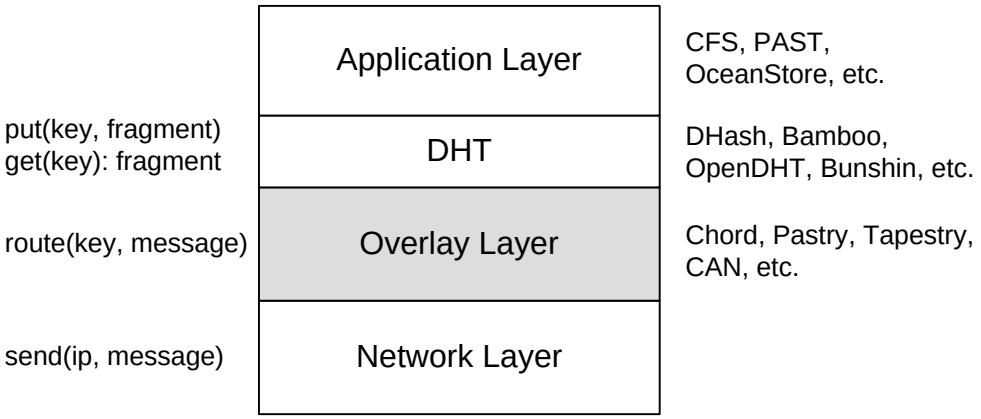
The advantages of peer-to-peer systems come at the price of higher algorithmic complexity. To tackle part of this complexity, good design solutions need to be applied. In this section, I will have a look at possible design issues when developing a peer-to-peer application. When talking about design, it is important to note that I am referring to software design (e.g. the level of classes and their relationships), as opposed to system design (e.g. routing algorithm, load balancing, and fairness).

In this paper, I am focusing on the overlay network, the common abstraction of most peer-to-peer systems. The overlay network is responsible for implementing the routing algorithm, which is the only function it offers to an application built on top. It uses the underlying network layer to send and receive messages over the network. The software architecture of a peer-to-peer application typically looks as follows.



As depicted, different kinds of applications can be built on top of the overlay network layer, such as file sharing applications (e.g. [55]), collaboration applications (e.g. [11], [12]), or distributed file systems (e.g. [28], [29], [31]). Distributed file systems, however, are usually not directly built on top of the overlay network layer, but on top of a distributed hash table (e.g. [42], [53], [56], [57]), which provides a higher abstraction in that it offers operations such as *put* and *get* to store

and retrieve data fragments in the network. In the literature, distributed hash tables and overlay networks are sometimes used interchangeably, because the major part of distributed hash tables are overlay networks, which are used to localize the fragments in the network. However, in this paper, I make a sharper distinction between those two terms, and regard distributed hash tables as a thin layer on top of the overlay network, as shown in the following figure.



Whether the peer-to-peer system is 'pure' in a sense that there is no central server involved at all, or if some aspects are still based on a central authority, does not matter from this point of view. All that is assumed is that the routing algorithm is implemented decentralized (as opposed for instance to the server-based approach in Napster). Authentication, for instance, might still be done using a central server, but those parameters are either not important for the overlay network, or simply fed in on initialization.

Following is a list of possible design problems that might occur in the overlay network. This list is not comprehensive and additional issues will arise in specific contexts. The patterns presented in this paper are trying to solve some of these problems.

- **Interaction with the application:** The application built on top of the overlay network uses it to route messages to specified keys (e.g. a query message to find a certain item in the network). How can the overlay network layer be completely encapsulated, so that it is not dependent on the application and can be used for different kinds of applications? How will the overlay network be accessed? When a message arrives at its target node, it needs to be delivered up to the application. How does the overlay network pass messages to the application? How does it notify the application of important events, such as for instance when it routes a message further to the next

node on the routing path? Some of these problems are known from other distributed systems, or even software systems in general, in which case the solutions simply need to be adapted to the requirements of overlay networks.

- **Routing:** Query messages need to be routed towards their target. On their way, they pass a lot of intermediate nodes. How are intermediate nodes processing the message? How does the message dispatching mechanism interact with the routing algorithm? Which objects are taking part in the routing process? How is the routing algorithm implemented, and how does it interact with the routing table and other local information available? Sometimes, not only messages sent by the application need to be routed, but others as well (e.g. join messages). How can different messages be made routable?

- **Direct messages:** Not all messages need to be routed, but some are sent directly to other nodes to maintain the routing topology (e.g. 'alive' messages). However, not only the overlay protocol needs to send direct messages, but also the application built on top. This is the case once an item has been found, and direct communication can take place. Unfortunately, this causes some design problems. Who is responsible to send these direct messages? Is it the overlay network, even though its task is only to *route* messages? Or is it the underlying network layer directly? But then, the application is built on top of two separate layers, which causes problems for instance when dispatching messages. Moreover, common abstractions cannot be shared anymore, and some code duplication will be necessary.

- **Roles:** Research networks often only know one type of node, which is responsible for all the different tasks. In real systems, however, this is rarely feasible, but different roles for the nodes need to be introduced. The reason for this is that the computers are highly heterogeneous. One example is that some machines are behind firewalls or network address translators (NAT), which makes it impossible to send unsolicited messages to such computers. These nodes can therefore sometimes not take part in the routing process. Another example are the differences in the resources of the individual computers (e.g. bandwidth, storage), so that some nodes can take more responsibilities to improve overall efficiency. How is this different behavior implemented in the overlay network? Even though different computers may be assigned different tasks, the source code should still be the same. On the other hand, 'symmetric' code is hard to read and maintain. How is this problem solved in the design of the overlay network?

- **Firewall, NAT:** Real systems often face the problem that some nodes are behind firewalls or network address translators (NAT), which renders it im-

possible to send unsolicited messages to such nodes. How is this problem solved in practical peer-to-peer systems? Are there reusable design solutions involved?

- **Message integrity:** Peer-to-peer systems are very exposed to the outside world. Because they communicate by exchanging messages, one simple form of attack is sending forged messages. How are such attacks detected and damage prevented? How can the message integrity be verified? Again, are there reusable design solutions involved?

- **Message hierarchy:** Lots of different kinds of messages exist in the overlay network. There are basic messages sent by the application, and control messages which are sent by the overlay network itself. Some of the messages need to be routed. Some need to be verified. How is the message hierarchy organized?

- **Message dispatching:** A message that arrives in the overlay network needs to be dispatched to the appropriate object which is responsible for processing it. Of course, this is true for all distributed systems. However, overlay networks represent a special case; some messages need to be processed in the overlay network, some need to be delivered up to the application, and some need to be sent away immediately to the next node. How is the interaction of the dispatcher with the routing algorithm? Which messages need to be delivered up to the application? How can the message processing be implemented clearly and readable?

- **Local node:** A node in the overlay network is often represented by a physical computer. Therefore, the notion of an explicit node is not necessary when designing the object model of the overlay network. What entities need to be present in the overlay network? How are local nodes included? In real systems, virtual nodes can be introduced as a simple way to introduce load balancing. The computer then hosts a number of nodes, rather than representing a node directly. However, separating the virtual nodes too strictly is sometimes not favorable because they can benefit from shared resources such as the routing table.

- **Protocols:** In order to maintain the routing topology of the overlay network, different protocols need to be executed periodically. Where and how are these protocols implemented in the code?

- **Remote nodes:** In the overlay network, the local nodes communicate with other nodes. In existing projects, it is often hard to identify the nodes or peers in the network. Instead of a clear abstraction for local and remote

nodes, these entities are obfuscated in the code. This makes it hard to read, understand and change the code. How are these remote nodes represented? How are messages sent to them?

- **Execution flow:** Overlay networks receive messages from other nodes, which either need to be routed further, processed by the overlay network, or delivered up to the application. The application in turn asks to route messages, and periodically, some protocol entity needs to send messages by its own. Furthermore, some events might occur which result in a notification of the application built on top. Altogether, a peer-to-peer application is highly concurrent, and good design practices need to be applied in order not to obfuscate the execution flow.

- **Testing:** Testing peer-to-peer systems is not easy, because setting up the environment takes a lot of efforts. For small changes, it should be possible to test the overlay network even without deploying the code to different computers and starting a test.

- **Dynamics:** A peer-to-peer system is inherently very dynamic. Computers can be turned on and off at any time, which results in nodes constantly joining and leaving the system. This makes it necessary to maintain and update the routing topology. If computers crash, they cannot properly leave the system. Still, the system needs to cope with these failures. How are these dynamics manifested in the code? Are there any reusable design solutions?

6 Pattern Language

This section presents a pattern language for overlay networks, the common abstraction of most peer-to-peer systems. The patterns are all on the conceptual level of overlay networks, dealing with entities such as messages, dispatchers, routers, nodes, and the like. This implies that a running network layer to transmit data over the network in a scalable way is presumed.

The patterns presented are trying to solve some of the design issues described in section 5. Most of the patterns are not new, however, but are either standard object-oriented design solutions or well-known patterns adapted to the specific needs of overlay networks. The remaining patterns are new suggestions that represent best-practices that have led to a favorable design in different projects. They can therefore be regarded as potential proto-patterns.

The goal of the pattern language is to collect and categorize expertise and knowledge in order to help software designers new to peer-to-peer systems, as well as experts, who find their ideas generalized in the form of patterns. Combining the patterns appropriately will result in a functional skeleton of an overlay network. Section 8 provides an example by putting together some of these patterns.

The rest of this section is organized as follows. Section 6.1 describes the form used to describe each pattern. In section 6.2, the patterns and their relationships are illustrated, and section 6.3 provides an overview over the problems and solutions that each pattern addresses, and of what type it is, either representing a standard design solution, a well-known, existing pattern, an adaptation from a well-known pattern, or a suggested proto-pattern.

6.1 Pattern Form

Patterns are written in various forms, such as the Alexandrian or canonical form, or the 'Gang Of Four' form [58], [59]. Important is not which exact form is used, but that certain important elements are described for each pattern. I closely followed the rules given in [60], which defines indispensable guidelines for pattern authors. In the end, I used the following form to describe the patterns, which is very similar to the form given in [60]. Note that not all elements need to be

present in all patterns.

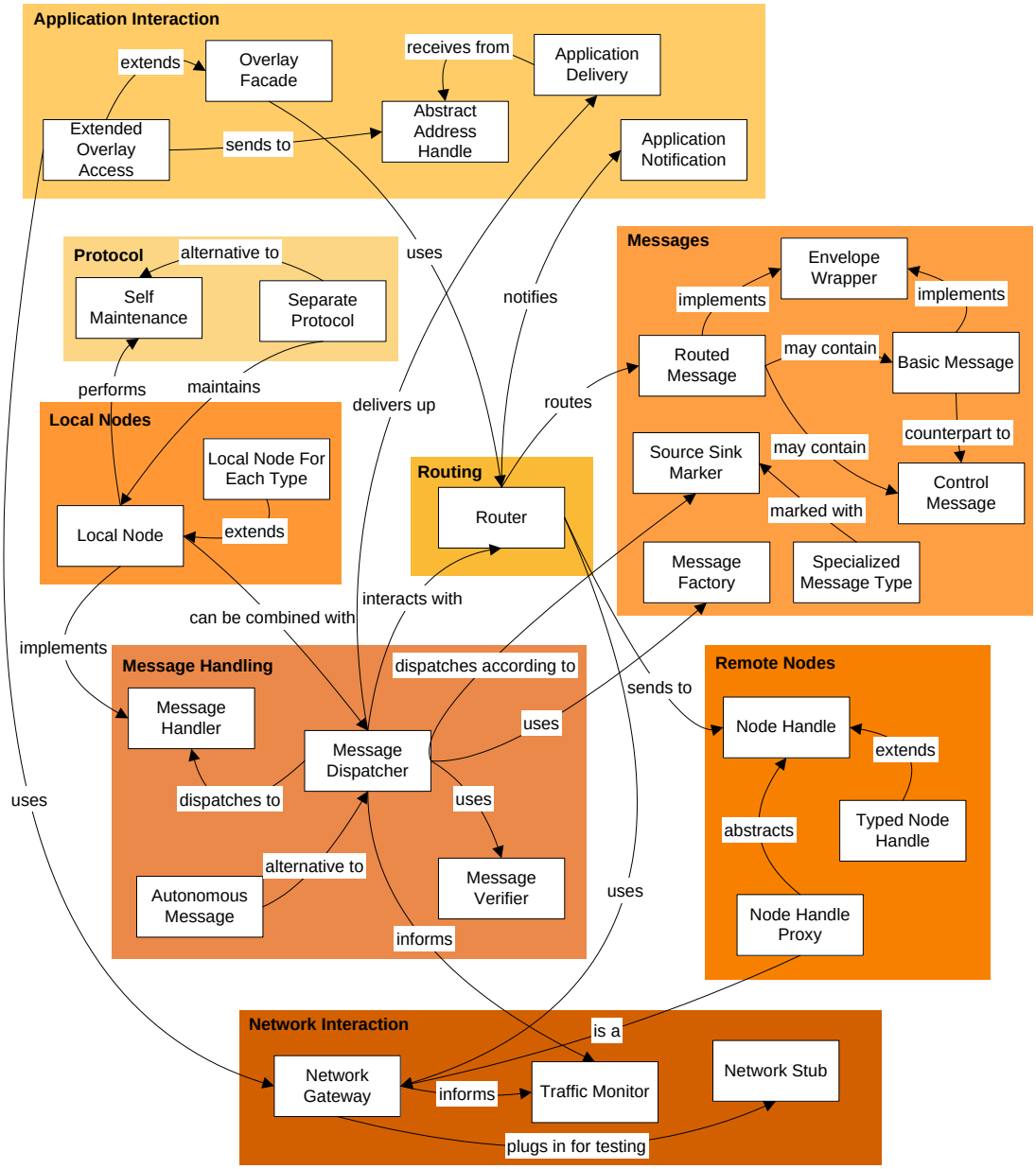
- **Pattern Name:** The name of the pattern.
- **Context:** The situation in which the problem occurs. If the pattern presumes the application of other patterns, they are stated here.
- **Problem:** The specific problem that needs to be solved.
- **Forces:** The considerations that need to be taken into account by the solution. Sometimes, the forces are contradictory (trade-offs), in which case not all forces can be resolved by the solution.
- **Solution:** The proposed solution to the problem in the given context, which takes certain forces into account. Sometimes, a UML diagram is used to illustrate the solution. However, these diagrams do not fully conform to the UML standard, but are rather used to sketch the idea in the simplest way.
- **Code Samples:** Some code samples to illustrate the solution in more detail. When code samples are provided, they are written in Java syntax. However, it should be no problem to convert them to another object-oriented language, such as C#.
- **Resulting Context:** The context that results after applying this pattern. It describes which patterns could be applied next.
- **Rationale:** An explanation of why this solution is most appropriate for the stated problem in this context.
- **Known Uses:** Concrete examples of where the pattern is applied.
- **Alternative Patterns:** Alternative patterns that can be used instead for a very similar problem.
- **Type:** The type of the pattern, representing either a standard design solution (S), a well-known, existing pattern (E), an adaptation from a well-known pattern (A), or a suggested proto-pattern (P).
- **References:** References to the literature or other patterns that have led to that solution, or that build the basis of the specific solution.

6.2 Overview

I have grouped the patterns in this language into seven categories:

- **Application Interaction** contains patterns about how the application interacts with the overlay network layer.
- **Messages** contains all patterns concerning messages.
- **Message Handling** describes patterns about how messages are processed.
- **Routing** describes how messages are routed towards their target.
- **Local Nodes** contains patterns about the concept of local nodes, including virtual nodes.
- **Remote Nodes** describes how remote nodes in the network can be represented.
- **Network Interaction** contains pattern about how the network layer could be accessed.

The following figure provides an overview over all patterns in this pattern language. The boxes represent the patterns, and the arrows indicate (some of) the relationships between the patterns. The big colored rectangles represent the different categories.


Application Interaction
extends
Overlay Facade
receives from
Application Delivery
Extended Overlay Access
sends to
Abstract Address Handle
Application Notification
uses
notifies
Protocol
alternative to
Self Maintenance
Separate Protocol
performs
maintains
delivers up
routes
Messages
Envelope Wrapper
implements
implements
Routed Message
may contain
Basic Message
counterpart to
Source Sink Marker
may contain
Control Message
marked with
Message Factory
Specialized Message Type
Local Nodes
Local Node For Each Type
Local Node
extends
Routing
Router
interacts with
can be combined with
implements
Message Handling
dispatches according to
Message Handler
Message Dispatcher
uses
sends to
Remote Nodes
Node Handle
extends
Typed Node Handle
abstracts
Node Handle Proxy
uses
dispatches to
uses
alternative to
Autonomous Message
Message Verifier
uses
informs
Network Interaction
is a
Network Gateway
informs
Traffic Monitor
Network Stub
plugs in for testing

6.3 Summaries

This section summarizes the pattern language by providing an overview over all problem / solution pairs. The second column indicates the type of each pattern: S = standard object-oriented design solution; E = well-known, existing pattern; A = adaptation from a well-known pattern; and P = suggested proto-pattern. In case the pattern already exists (E) or is an adapatation from a well-known pattern (A), the original pattern is stated in the name column in brackets.

Application Interaction

Pattern		Problem	Solution
Overlay Facade (Facade)	E	How do you encapsulate access to the overlay network from the application?	Use an Overlay Facade, which exposes the operations that the overlay network provides to the application (API), and encapsulates the implementation of the overlay network.
Application Delivery (Observer)	E	How does the overlay network deliver up messages to the application?	Use an Application Delivery interface with a *deliver()* method, which allows the overlay network to deliver up messages by simply calling this method. An object implementing this interface is provided by the application.
Application Notification (Observer)	E	How do you notify the application in case of an important event?	Use an Application Notification interface with known methods to the overlay network, so that it can inform the application simply by calling the appropriate method.
Extended Overlay Facade	P	How are direct messages sent? Are they sent using the overlay network, or by accessing the underlying network layer directly?	Extend the Overlay Facade with a *send()* method, and use the overlay network layer also to send direct messages to known addresses.
Abstract Address Handle	P	How do you encapsulate the necessary algorithm to contact computers behind firewalls or network address translators (NAT), without affecting the application built on top?	Use an Abstract Address Handle to refer to another computer. The Abstract Address Handle encapsulates all information necessary to contact computers even if they are behind firewalls or NATs. However, for the application, they can be used as if they were Internet addresses.

Messages

Pattern		Problem	Solution
Message Factory (Factory)	E	How do you transform the raw bytes into the different message objects?	Use a Message Factory, which provides a *create()* method that takes the raw bytes as input and returns the appropriate message object. The Message Factory encapsulates the logic to transform the bytes into the message objects properly.
Envelope Wrapper (Envelope Wrapper)	E	How do you send a given message with another messaging system?	Use an Envelope Wrapper to wrap the message to be sent in an envelope that is compliant with the message system used to send the message.
Basic Message (Envelope Wrapper)	E	How do you separate messages sent by the application from other messages sent by the overlay network?	Use Basic Messages to wrap messages sent by the application in a simple Envelope Wrapper. A Basic Message can be as simple as only containing the payload. However, it allows the Message Dispatcher to distinguish it from Control Messages sent by the overlay network.
Control Message	S	How do you structure control messages?	Use Control Message as a common super type of all control messages. This structures the message hierarchy clearly into Basic Messages and Control Messages. Additionally, common properties and behavior can be defined in the Control Message super type.
Routed Message (Envelope Wrapper)	A	How do you make specific messages routable?	Use a Routed Message which wraps the message to be routed and adds the necessary header fields that are important for the routing algorithm.
Specialized Message Type	S	How can you improve testability on the messages and profit from static type-safety to render some faulty network conditions impossible?	Use Specialized Message Types, which allows monitoring exactly which messages two nodes exchange. It also helps for debugging and allows each message to be adapted to the specific needs of each pair of node types, thus including strong types for the fields. Wrong assignments can therefore already be checked by the compiler.
Source Sink Marker (Marker)	E	How can you make the source and sink node type of a message explicit, so that it can be checked in the code?	Use Source Sink Marker to mark the source and sink node type of each message. Let each message simply implement this (empty) Marker Interface, so that the node types can be checked programmatically.

Message Handling

Pattern		Problem	Solution
Message Dispatcher (Observer)	E	How do you process the different messages?	Use a Message Dispatcher, which receives the raw bytes from the network layer, creates the corresponding message object using the Message Factory, and dispatches it to the actual object responsible for processing it.
Message Handler (Observer)	E	How can you make the Message Dispatcher be as simple and small as possible?	Use a Message Handler interface which simply provides a method *handle(Message message)*. All objects that are responsible for processing messages need to implement this interface. Then, the Message Dispatcher can call this method to dispatch off messages and is prevented from processing them by itself.
Autonomous Message (Command Message)	A	How could you let messages process themselves, making the Message Dispatcher and Message Handlers become redundant?	Use Autonomous Messages, which know how to process themselves. They are processed by calling an *execute()* method on the message object.
Message Verifier	P	How can you include a simple verification mechanism, which can be extended for specific messages?	Use a Message Verifier, which provides a method *verify()* that checks the integrity of all messages. If credentials are included, it will verify that they are indeed issued for the claimed sender.

Routing

Pattern		Problem	Solution
Router	P	How do you implement the routing algorithm and how does it interact with other objects in the overlay network?	Use a Router, which encapsulates the routing algorithm and provides a method to route Routed Messages. This method checks whether a Routed Message is at its target, or if it needs to be sent away, in which case the router chooses the next node according to its routing algorithm from the routing table and neighbors table. It interacts with the Message Dispatcher to dispatch messages that have arrived at their target to the appropriate object.

Local Nodes

Pattern		Problem	Solution
Local Node	S	How do you model the object space of the overlay network around the concept of local nodes?	Use a Local Node for each node that is hosted on the computer, resulting in a visual and clear structure in the object space. The Local Node has its own identifier and stores connections to remote nodes (Node Handles), which represent the topology of the overlay network.
Local Node For Each Type	S	How do you integrate different node types with different behavior?	Use Local Node For Each Type, which represents each different node type in the network explicitly. Local Node For Each Type extends the Local Node, which provides the properties and behavior that are shared among all node types.

Protocol

Pattern		Problem	Solution
Self Maintenance	P	How do you implement the maintenance protocol?	Use Self Maintenance, which encapsulates the maintenance protocol in the Local Node and runs it periodically in its own thread. This makes the nodes be the only active components in the system, responsible for joining and maintaining themselves the same way as in the system model.
Separate Protocol	S	How do you implement different protocols, needing to run at different time intervals?	Use a Separate Protocol for each different protocol, encapsulating the respective logic. Each Separate Protocol can be run at different time intervals.

Remote Nodes

Pattern		Problem	Solution
Node Handle	S	How do you store all the different information available about a remote node in the overlay network?	Use a Node Handle, which provides an abstract handle of a remote node. It stores all available information about a node at a central place. Make the Node Handle serializable or write a proprietary marshalling algorithm, so that the necessary information can be transmitted easily.

Pattern		Problem	Solution
Typed Node Handle	S	How can you make the Node Handles 'type-safe', making it easier to detect mistakes and improve readability?	Use a Typed Node Handle, which is simply an extension of a Node Handle for each type of node. Make the base type abstract and consequently use the appropriate Typed Node Handle throughout the overlay network and in message objects.
Node Handle Proxy (Proxy)	A	How can you refer to nodes in the same way, whether they reside locally or remotely, thus making the underlying transmission of a message transparently?	Use a Node Handle Proxy, which can represent both, a remote or a local node. It provides a method *receive(Message)* which either lets the Local Node process the message or sends the message using the underlying network to the remote node.

Network Interaction

Pattern		Problem	Solution
Network Gateway (Gateway)	E	How do you remove strong dependence on the network layer and allow to put in control mechanisms for outgoing messages?	Use a Network Gateway, which encapsulates access to the underlying network gateway.
Network Stub (Service Stub)	E	How can you include a simulation environment for your overlay network?	Use a Network Stub, which uses the same interface as the network layer, but behaves differently, for instance simulating the sending and receiving of messages.
Traffic Monitor	S	How can all incoming and outgoing messages easily be monitored?	Use a Traffic Monitor at the overlay network layer, which is informed of all incoming and outgoing messages, interprets them and updates its statistics.

7 Patterns

7.1 Application Interaction

This section contains patterns that deal about the interaction of the overlay network with an application built on top of it. Most of them are adaptations from well-known patterns from other layered or component-oriented software systems.

7.1.1 Overlay Facade

Context

You are developing an overlay network, which solves the task of routing a message to a given key. You want to expose this functionality to an application, such as a distributed hash table, built on top of your overlay network.

Problem

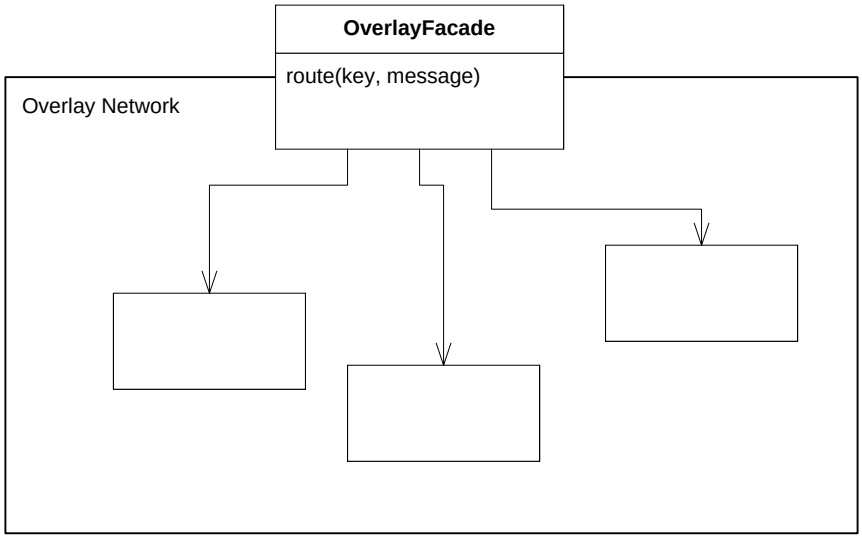
How do you encapsulate access to the overlay network from the application?

Forces

- The overlay network should be completely encapsulated, so that changes in the implementation do not affect the application at all.
- The overlay network should not depend on the application built on top, so that it can be reused by different applications.
- The access to the overlay network should be as clear and simple as possible. The underlying complexity should be completely hidden, making it trivial to use the overlay network to route messages to a specified key.
- The application should not be dependent on the overlay network either, so that the overlay network can simply be replaced by another one.
- Accessing different parts of the overlay network directly can be more efficient.

- It should not be possible to use the overlay network in the wrong way.

Solution



Use an Overlay Facade, which exposes the operations that the overlay network provides to the application (API), and encapsulates the implementation of the overlay network.

Specify the operations of the overlay network in an Overlay Facade. The implementation then accesses the appropriate objects of the overlay network to implement the operations provided. In order to initialize the overlay network, only the Overlay Facade object needs to be constructed. This will be the only reference that the application will have into the overlay network layer.

The methods specified in the Overlay Facade depend on what the overlay network layer provides to the application. In most cases, this will be a *route()* method, which routes messages to a given key. The interface corresponds to the API that the overlay network layer offers to any application built on top of it.

The actual implementation of the Overlay Facade pattern can look differently. Whether the Overlay Facade object is created directly, or whether an initialization class is used that returns an Overlay Facade interface, depends on the specific initialization procedure.

Resulting Context

Using Overlay Facade, the application can now access the functionality provided by the overlay network. The overlay network, on the other hand, does not have a reference to the application, which is needed for instance to deliver up messages that arrive for that application. For that reason, you will need Application Delivery.

Additionally, you might want to use the overlay network not only to route messages, but also to send them directly to a known address, once the query message has been answered. In that case, Extended Overlay Facade can be used instead.

The application can implement a Gateway, as well as a Service Stub, on top of the Overlay Facade as is done with Network Gateway and Network Stub to access the network (see [37]). This allows the application to test itself even without a proper overlay network in place.

Rationale

The solution is very simple, but solves the problem effectively. The same solution is also applied in other layered and component-oriented software systems (see References). It resolves most forces stated:

- Because the application can only access the overlay network through the operations provided in the Overlay Facade, the implementation can be freely changed and does not affect the application, as long as it remains semantically equivalent.
- The overlay network does not depend on the application built on top at all, so that any application can make use of it.
- Because of the narrow interface provided by the Overlay Facade, the application does not depend that much on the specific overlay network. This makes it trivial to switch to another overlay network that provides the same functionality.
- The Overlay Facade makes it very simple to use the overlay network. The underlying complexity is completely hidden.
- Accessing the objects in the overlay network directly would be slightly more efficient, because the Overlay Facade needs to delegate the calls to the appropriate hidden object. However, in most cases, this effect is negligible, because it is orders of magnitudes smaller than the resulting waiting time due to the message transport.

- Because of the restricted access, the Overlay Facade makes it impossible to use the overlay network in the wrong way.

The solution has the following favorable qualities:

- Encapsulation: The Overlay Facade encapsulates the overlay network as a component completely.
- Reusability: The overlay network as a component can easily be reused by another application, because the Overlay Facade makes an explicit interface.
- Low coupling: Because of the single access point, using Overlay Facade will result in low coupling between the application and the overlay network.
- High cohesion: The explicitly stated operations will make the component as a whole very cohesive.
- Understandability: Overlay Facade makes it trivial to understand the functions provided by the overlay network. From a high-level point of view, it is not necessary to jump into the overlay network when reading the source code.
- Simplicity: Using the overlay network is very simple due to the explicit operations provided by the Overlay Facade.
- Clarity: The Overlay Facade results in a clear structure and a clear access point of the overlay network component.
- Modularity: The Overlay Facade encapsulates the overlay network component as a module, with the only external methods specified in the Overlay Facade.
- Abstraction: Overlay Facade is a nice abstraction to specify the external interface of the overlay network.
- Extendibility: The overlay network can be extended freely and the functionality can always be added to the Overlay Facade.

Known Uses

Facades are used in a lot of software system, when access to another component needs to be encapsulated or simplified. In overlay networks, the Facade pattern is also the preferred pattern for this need. FreePastry combines the Overlay Facade with Application Delivery and Application Notification in an application object.

Type

Existing pattern (E): Facade.

References

Overlay Facade corresponds to the well-known Facade pattern, described in 'Design Patterns' [19]. In the Facade pattern, it is possible that the Facade simply provides a simplified interface, but that other objects in the subsystem can still be accessed directly. In this case, we want to restrict access completely, so that not other objects can be accessed by the application at all.

Overlay Facade is also related to the Gateway pattern, described in 'Patterns of Enterprise Application Architecture' [37]. Compared to the Gateway pattern, however, the Facade is provided by the subsystem, whereas a Gateway is mostly written by the programmer accessing the subsystem (see [37] or Network Gateway) from the outside.

There are a number of other patterns closely related to this. Wrapper Facade is most often used to wrap a non-OO API, which is not the case for this pattern. Adapter is also related, however, in this case, the Overlay Facade does not try to adapt an existing interface, but creates its own. The Handle Body pattern can be used to implement this pattern, separating the interface from its implementation. A more generic variant of Handle Body is described by the Bridge pattern. Messaging Gateway is also very similar, in that it provides an access point to a messaging system (see [35]).

The methods that the Overlay Facade provides can conform to the API specified in [27].

The creation of an overlay socket (see [32]) is very similar to the creation of an Overlay Facade.

7.1.2 Application Delivery

Context

The routing functionality, which the overlay network provides to the application, is inherently asynchronous. When using the *route()* operation to route a message to a key, the message is simply routed towards its target, but no answer is given as a return value of the operation. This implies that an answer, once it arrives, needs to be delivered up to the application. Moreover, at the target node, the query message needs to be delivered up to the application, without this application having used any operation provided by the overlay network, thus asynchronously.

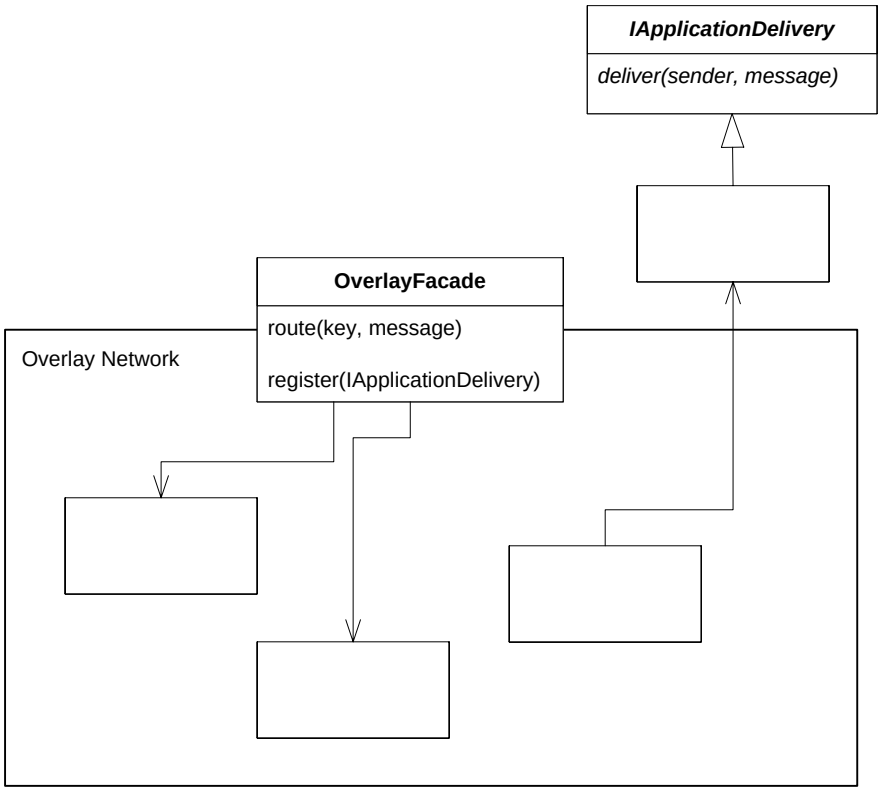
Problem

How does the overlay network deliver up messages to the application?

Forces

- The overlay network should be completely encapsulated and reusable for other applications. An explicit reference to an object in the application layer is therefore not a solution.
- Making the operations provided by the Gateway synchronous and therefore avoiding upcalls is not feasible for a number of reasons. An obvious one is that messages that arrive at the responsible node need to be delivered up without this application having used any operation at all.
- More than one application might be using the same overlay network at the same time.

Solution



Use an Application Delivery interface with a *deliver()* method, which allows the overlay network to deliver up messages by simply calling this method. An object implementing this interface is provided by the application.

The Application Delivery interface is defined by the overlay network, but the application is responsible for implementing it. The interface will need to contain a method similar to *deliver(Address sender, Message message)*. When messages arrive that need to be delivered up to the application, the overlay network will call this method to deliver them.

Since the message cannot be interpreted by the overlay network, it will deliver either the raw bytes, or a common message object known to the overlay network as well as the application (the same common message object can be used in the Overlay Facade). *Sender* is the address of the original sender of this message. The sender address might be either an Internet address, or an Abstract Address Handle.

If only one application can be built on top of the overlay network, then the Application Delivery object can simply be specified when initializing the overlay network layer (see Code Samples). If several applications should be built on the same overlay network instance, the overlay network needs to store a list of all objects conforming to Application Delivery, instead of only one. There might be a *register(IApplicationDelivery applicationDelivery)* method in the Overlay Facade (see UML diagram). Because it does not know to which application a message should be delivered, it will need to deliver it to all applications and let it be their responsibility to decide. However, if more flexible mechanisms are needed, it would be easy to use an application type identifier to let the overlay network know to which application to deliver to.

Resulting Context

Using Application Delivery, the overlay network knows how to deliver up messages to the application. How the application accesses the overlay network is explained in Overlay Facade. The sender object can either be an Internet address, or an Abstract Address Handle.

Besides messages, there might occur other events in which case the overlay network might need to inform the application. For that case, use the Application Notification pattern.

Rationale

The Application Delivery is a simple mechanism to let the overlay network communicate with the application, even though it does not know it beforehand. It

resolves all forces stated:

- It allows building any application on top of the overlay network, and makes it therefore completely reusable.
- By providing an Application Delivery interface to the overlay network, the operations can be made asynchronous.
- It even allows building multiple applications on top of the same overlay network instance, by storing a list of Application Delivery objects in the overlay network.

The solution has the following favorable qualities:

- Low coupling: Using Application Delivery results in minimal coupling between the overlay network and the application built on top, because the overlay network does not have an explicit reference to the application.
- Flexibility: Because of the interface provided by Application Delivery, any kind of application can be built on top of the overlay network. Furthermore, it is also possible to register several applications at the same time for a single overlay network.
- Clarity: Application Delivery makes it very clear where and when the overlay network delivers up messages to the application.
- Understandability: It is very easy to understand the message flow between the overlay network and the application built on top when using Application Delivery, because the methods that exchange data are explicitly stated.

Known Uses

Most overlay networks today are designed for reuse. Therefore, they all have similar mechanisms to register an application not known beforehand with the overlay network. The listener pattern is predestined for this purpose and is used in most systems. FreePastry, for example, has been used for different kinds of applications such as PAST [29], Scribe [30], and others [54]. The JXTA framework is very flexible, and the listener pattern is used throughout the design.

Alternative Patterns

The approach taken in the Berkeley socket API provides an alternative solution to handle asynchronous messages. There, the application uses the blocking *receive()* to receive messages. This has at least two advantages. First, the application can call *receive()* whenever it is ready to process the next message. Second, the processing is done by another thread than the one who delivers up the messages. The latter advantage is especially important, depending on the system design and whether the application executes a lot of (potentially also blocking) methods. However, this can also be controlled by the overlay network which can start a new thread for the delivery. Using a *receive()* would need a buffer in the overlay network to cache messages that have not yet been delivered up. In any case, concurrency issues need to be solved and the application must be carefully designed (see for instance [34]).

The overlay socket API (see [32] uses this approach for delivering messages. P2P Sockets [61] is a project that provides a socket based API on top of JXTA.

In case Application Notification needs to be applied, Application Delivery could be the better alternative to deliver up messages, because the two mechanisms are very similar.

Type

Existing pattern (E): Observer.

References

Application Delivery is very closely related to the well-known Observer pattern [19]. There, the Application Delivery object corresponds to the observer. Application Delivery is related to event listeners in general, which are used in event systems (e.g. GUI frameworks).

The Application Delivery interface can conform to the common API defined in [27].

7.1.3 Application Notification

Context

A number of events occurring in the overlay network may be of interest to an application built on top. One such event is for example defined in [27]; when the overlay network is about to route a message to the next node, a *forward()* method in the application should be called. If this method returns *true*, the message can be sent off, otherwise, it needs to be dropped. This allows the application to control

whether a message should be routed further. Other possible events include *ready()* or *disconnected()*, in which case the application might have more information on how to bootstrap again, for instance by providing some new seed nodes.

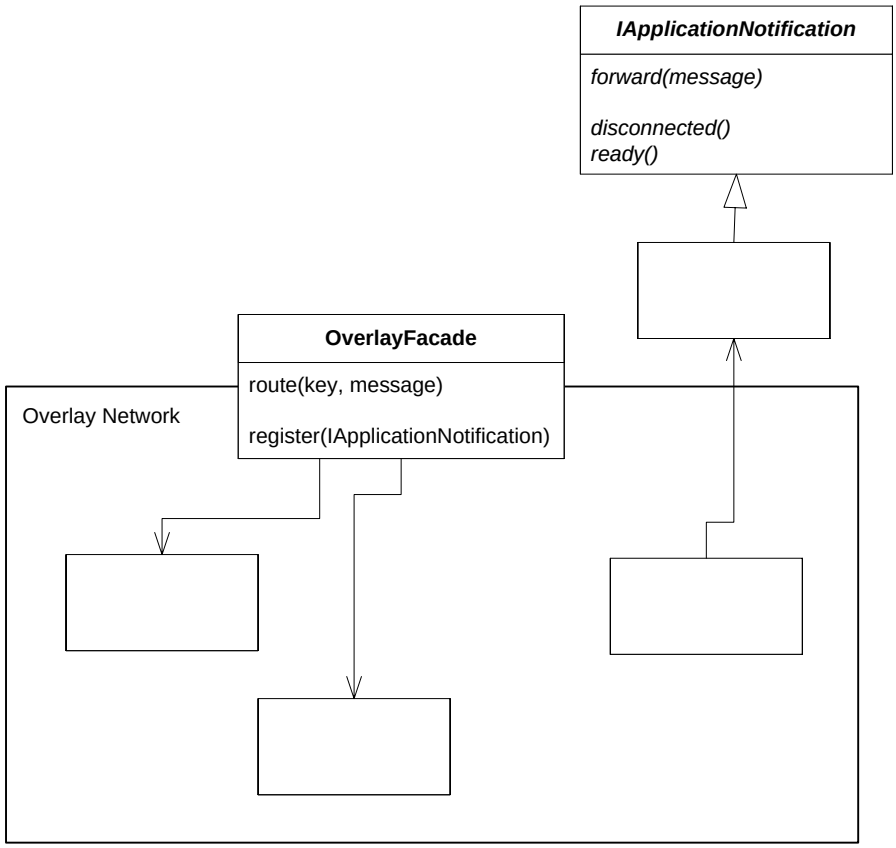
Problem

How do you notify the application in case of an important event?

Forces

- The overlay network should be completely encapsulated and reusable for other applications. An explicit reference to an object in the application layer is therefore not a reusable solution.
- Different events might occur.
- An actual application might not be interested in some or all of the events.

Solution



Use an Application Notification interface with known methods to the overlay network, so that it can inform the application simply by calling the appropriate method.

This solution is very similar to Application Delivery. The application needs to implement an interface, defined in the overlay network layer, which gives the overlay network known methods to call in case an event happens.

In the simplest case, the interface can for instance provide methods such as *ready()*, *disconnected()*, *forward(Message message)* to inform the application of the respective event. In most cases, such simple events are sufficient (see also [27]). If a more flexible and generic solution is needed, a proper event listening mechanism can be applied, where an event object would be passed as a parameter. The event object would inform the application of the respective event. Such a solution is more flexible and allows adding new event types easily. Furthermore, applications could subscribe only to those events that are interesting for them.

Resulting Context

Once Application Notification is applied, the application can be informed of events happening in the overlay network. In the case of the *forward()* method, this even gives some control to the application, which can now decide whether a message may be routed further.

Rationale

Using Application Notification is very simple and similar to Application Delivery. It resolves all forces stated, and can be extended whenever the need arises:

- The overlay network is not dependent on the application, because the Application Notification interface is defined in the overlay network.
- In the simplest case, different methods for different events can be specified in the interface. If a more flexible mechanism is needed, the proper event can be passed as a parameter in an event object.
- In the simplest case, the application implements the methods specified in the interface, but does not do anything in it (empty method). Again, this can be made more flexible by implementing a proper event listening mechanism, where an application can subscribe to individual events only. However, in most cases, this is not necessary.

The solution has the following favorable qualities:

- Low coupling: Similar to Application Delivery, also Application Notification results in minimal coupling between the overlay network and the application built on top, because the overlay network does not have an explicit reference to the application.
- Flexibility: Application Notification is a very flexible mechanism, because the application can decide which events it is interested in and how it reacts.
- Extendibility: Adding a new event to Application Notification is very easy.
- Clarity: Application Notification results in a clear and visual structure for the possible events of the overlay network.
- Understandability: Using Application Notification, it is easy to understand where and when an event is triggered.

- Not overdesigned: Application Notification can be extended as needed. In simple overlay networks with only few events that are known beforehand, it is not necessary to make the Application Notification mechanism very generic. Simply calling a method provided in the interface is sufficient. However, if an application should be able to register for certain specific events only, it is easy to change the implementation to a proper event listening mechanism.
- Documentation: The explicit Application Notification serves as an ideal documentation for all events that can be fired by the overlay network.

Known Uses

Again, as in Application Delivery, most overlay networks use the listener pattern to notify the application of events.

Type

Existing pattern (E): Observer.

References

Again, as in Application Delivery, Application Notification is very closely related to the Observer pattern and to event listeners in general. It is stated here as its own pattern simply because it is an architectural element of most overlay networks.

In [27], some possible events that could happen in the overlay network are defined.

7.1.4 Extended Overlay Facade

Context

The task of the overlay network is to route messages to given keys. Most often, messages that need to be routed are query messages, which try to localize an item in the network. Once a reply has been received, the application can now send further messages to the responsible node directly, and no routing is needed any more. Sending a message directly to a node, however, is not the task of the overlay network layer, but of the underlying network layer.

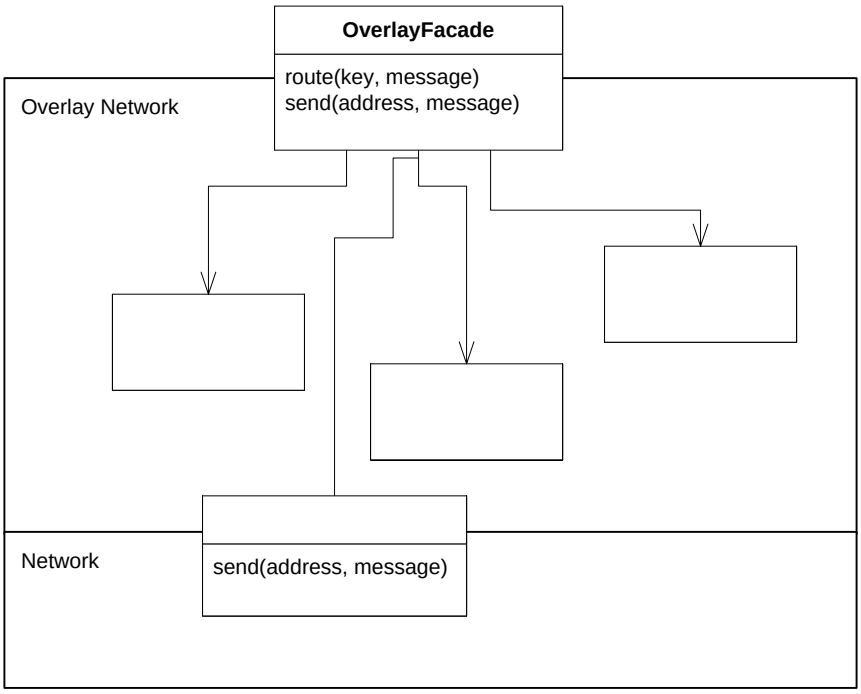
Problem

How are direct messages sent? Are they sent using the overlay network, or by accessing the underlying network layer directly?

Forces

- The overlay network layer's task is only to route messages to keys, not to send them directly to a given address (e.g. Internet address).
- The *send* operation to directly send a message to a given address is implemented in the network layer, thus it should not be duplicated in the overlay network.
- Having only one abstraction for communication, routing *or* sending, is much more convenient and simpler for the application built on top.
- Exposing the underlying network layer to the application does not conform to common layering principles, and creates more dependencies.
- Additionally, if the network layer is used to send messages directly, then this layer would also need to know whether to dispatch a message to the overlay network layer, or to the application layer. Thus, it would need to be able to interpret at least some part of the message as well.

Solution



Extend the Overlay Facade with a *send()* method, and use the overlay network layer also to send direct messages to known addresses.

Using Extended Overlay Facade, the overlay network completely hides the underlying network layer. The *send()* method simply delegates the call to the appropriate method of the network layer, using for instance Network Gateway. If the specified address is an Abstract Address Handle, the overlay network might even help to circumvent firewalls or network address translators (NAT).

Resulting Context

If the receiver of the message might potentially be behind a firewall or network address translator (NAT), use Abstract Address Handle instead of an Internet address as the destination of the message. The message to be sent should be wrapped in a Basic Message, to let the Message Dispatcher distinguish it from Control Messages at the receiving end. Then, the Message Dispatcher can deliver it to the application using Application Delivery. The Extended Overlay Facade should make use of a Network Gateway to send the message using the underlying network layer.

Rationale

Extended Overlay Face enhances the responsibilities of the overlay network layer. This extension is not necessarily favorable, because sending messages is not the actual task of the overlay network. It does, however, provide some significant advantages, which makes it the better alternative in most cases. First, the overlay network completely hides the underlying network layer. This respects common layering principles (e.g. [63]), which allows changing lower layers without affecting upper layers in a software system. Second, it avoids code duplication. Since applications have already properly registered themselves with the overlay network (using Application Delivery), they do not need to do the same with the network layer as well. Third, in some contexts, it does have a more important and practical reason as well. In case some computers are behind firewalls or network address translators (NAT), the overlay network may have some additional information to contact a computer. It may be possible to contact a certain computer over its connection to another node, which is not behind a firewall or NAT. When using Abstract Address Handle, the application does not have to worry about this and can simply send messages using Extended Overlay Facade to addresses (Abstract Address Handles) it has received from the overlay network (Application Delivery).

Extended Overlay Facade resolves most forces stated:

- The actual task of the overlay network is only to route messages. To reconcile the purist view, the overlay network's task can be extended to all communication tasks. Thus, if the overlay network layer is viewed as the only communication layer, the *send* operation neatly fits in.
- The overlay network does not duplicate the *send* operation, it simply delegates the call to the network layer.
- The Extended Overlay Facade makes it very simple for the application to route and send messages.
- Using the Extended Overlay Facade allows hiding the underlying network layer completely, thus following strict layering principles.
- If the application only connects with the overlay network layer, the same dispatcher and resources can be used.

The solution has the following favorable qualities:

- Encapsulation: The Extended Overlay Facade completely encapsulates the underlying network layer from the application.
- Low coupling: The application is completely decoupled from the network layer. Changes to the network layer do not affect the application at all.
- Avoids code duplication: Because the overlay network does need to know which messages to deliver to the application already, this code does not need to be duplicated for the network layer again.
- Simplicity: For the application, it is very simple to only need to communicate with the overlay network layer, instead of two different layers at the same time. The functionality provided in the Extended Overlay Facade is easy to understand.
- Layering principles: The Extended Overlay Facade results in good layering principles, because the application layer only needs to communicate with its lower layer, and the network layer is completely shielded away.

Unfortunately, the solution misses the following qualities:

- High cohesion: In a pure view, the *send* operation would belong to the network layer only.

- Abstraction: Again, the abstraction of the Extended Overlay Facade is somehow wrong, because the overlay network is not responsible to send simple direct messages.

However, these two points can be reconciled if one regards the overlay network layer as the only communication layer the application needs to talk to.

Known Uses

The issue addressed by this pattern is especially relevant to practical systems. Therefore, most practical systems provide a method to send messages directly to an address, once it is known. However, often the underlying network is used directly.

Type

Proto-pattern (P).

References

Of course, this pattern is a simple extension of the Overlay Facade. However, because it has a number of design consequences, it is stated as its own pattern. A *send* operation is also specified in [27].

The Overlay Facade could even be extended further to provide more methods to an application built on top. One could imagine statistical information of the overlay network which could be of interest to an application. However, because of the different semantics of these methods, a second interface for this could be provided, as is for instance done in [32] with the statistics interface.

7.1.5 Abstract Address Handle

Context

The overlay network's routing functionality is used to localize nodes in the network. Once a node has been found, further messages can be sent directly to its address. If you are using Extended Overlay Facade, then this *send()* method is provided by the overlay network. Unfortunately, in real networks, it is sometimes not possible to send a message to an Internet address directly, because of firewalls or network address translators (NAT). In that case, the communication must either be relayed over a well-configured node, or 'hole punching' techniques need to be applied. This renders it impossible to simply use an Internet address as a parameter of the *send()* method. However, the application built on top should not need to worry about

these problems and should be able to simply send messages to the return address provided by the overlay network in Application Delivery.

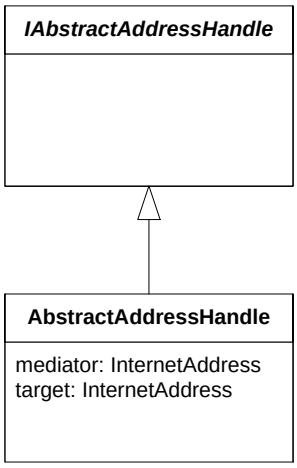
Problem

How do you encapsulate the necessary algorithm to contact computers behind firewalls or network address translators (NAT), without affecting the application built on top?

Forces

- The application should simply be able to send a direct message to the return address that it has received from the answer of a query (Application Delivery).
- Not all computers are well-configured in practical systems, making it impossible to simply use Internet addresses.
- Contacting a node hosted on a badly-configured computer may need the interaction of more than one node. The underlying complexity should be hidden from the application.

Solution



Use an Abstract Address Handle to refer to another computer. The Abstract Address Handle encapsulates all information necessary to contact computers even if they are behind firewalls or NATs. However, for the application, they can be

used as if they were Internet addresses.

The Abstract Address Handle simply serves as a handle to a remote node for the application. The application does not need to access the address object's internals. Therefore, all the application receives in Application Delivery is an empty interface with no methods, which serves as such an abstract handle. If this Abstract Address Handle interface is used in the Extended Overlay Facade, the overlay network can cast the interface to the appropriate object which provides access to all information necessary to contact that computer, including its Internet address and possibly a mediator Internet address.

How the firewall or NAT is traversed depends on the actual implementation. The mediator (relay) address contained in the Abstract Address Handle might relay all messages to that node, or it may be used to initiate 'hole punching' methods.

Resulting Context

If you are using Abstract Address Handle, then the receiver Internet Address in the Extended Overlay Facade can be replaced by Abstract Address Handle. Consequently, Application Delivery will deliver Abstract Address Handles instead of Internet addresses as well. This gives the application an abstract handle to communicate to a physical computer. In order to communicate, it will need the overlay network, and thus Extended Overlay Facade in place.

Rationale

Using an Abstract Address Handle provides an abstract handle for the application to refer to a remote computer, no matter whether it is behind a firewall or NAT. In case it is, the overlay network can read the mediator address out of the casted address object and can start the mechanism to contact the computer. If the computer is not behind a firewall or NAT, its Internet address can be used directly by the overlay network.

Abstract Address Handles resolves most forces stated:

- The application can simply send a message to the address it has received from the overlay network. The logic to contact nodes behind firewalls or NATs is completely encapsulated in the overlay network.
- Because not all computers can be contacted directly, the overlay network never returns simple Internet addresses, but Abstract Address Handles.

- Because the overlay network has the knowledge about the connection of nodes in the network, it is the correct place to implement the traversal logic. Abstract Address Handles hides the underlying complexity completely.

The solution is good because it has a number of favorable qualities:

- Abstraction: The Abstract Address Handle is a nice abstraction to refer to remote addresses, because it is completely transparent how the underlying layer contacts this address. Otherwise, the application would need to care about these issues.
- High cohesion: Abstract Address Handle puts the logic on how to contact a remote address at the right place, so that it results in a highly cohesive architecture.
- Simplicity: For the application, it is trivial to send a message to a given Abstract Address Handle.
- Understandability: Using Abstract Address Handle is very easy to understand, because the traversal logic is encapsulated and does not need to be understood.
- Encapsulation: The traversal logic is encapsulated, so that it can freely be modified without affecting the application.
- Flexibility: It is easy to support different lower layer transport protocols and the like, because the application does not need to take care about these issues at all.

Solution

Known Uses

A lot of academic overlay network do not take problems arising from firewalls or NATs into account, so that this pattern is not applied there. Practical projects sometimes use hole punching techniques, but are implemented rather ad-hoc, so that no nice abstractions can be found. The emtpy *Address* interface in FreePastry is a similar idea, even though its purpose is a little bit different.

Type

Proto-pattern (P).

7.2 Messages

Overlay networks communicate by exchanging messages. This section contains patterns describing solutions to all kinds of design problems with messages in the overlay networks. Of course, messages are an integral part of any distributed systems, and good design patterns for messages have been around for a long time. Therefore, some of these patterns are adapted from other messaging systems (e.g. 'Enterprise Integration Patterns'), and are listed in this pattern language only for the sake of completeness.

7.2.1 Message Factory

Context

Nodes communicate by exchanging messages. Serializing message objects and transmitting them by using high-level programming language abstractions is very convenient and is often the preferred method used in traditional distributed systems. For efficiency reasons, however, messages in peer-to-peer systems are sometimes exchanged by transmitting the raw bytes that make up a message directly, using lower-level protocols such as UDP. Therefore, the marshalling and demarshalling needs to be implemented proprietarily. On the receiving end, the network layer has no knowledge about the messages and simply delivers the raw bytes that correspond to one logical message up to the Message Dispatcher. These bytes then need to be transformed into the appropriate message object to further process it conveniently.

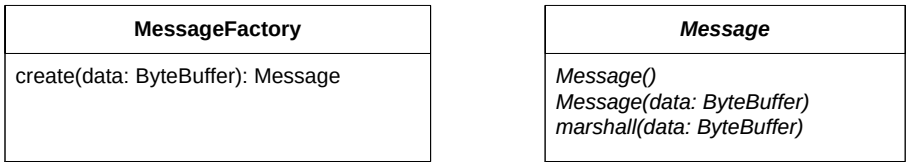
Problem

How do you transform the raw bytes into the different message objects?

Forces

- Processing message objects is much more convenient than processing raw bytes directly.
- The bytes enter the overlay network through the Message Dispatcher. However, knowing how to interpret these bytes is not the task of the Message Dispatcher.
- During the development process, adding new message types should be easy.
- Message objects may know how to marshall and demarshall themselves.
- Changing message types should not affect a lot of places in the source code.

Solution



Use a Message Factory, which provides a *create()* method that takes the raw bytes as input and returns the appropriate message object. The Message Factory encapsulates the logic to transform the bytes into the message objects properly.

The Message Factory itself does not need to know how to create the different messages. Instead, it lets the message object construct itself. However, the Message Factory knows how to read the type information from the bytes, which serves as an index in the message type table. It can then call the constructor of the appropriate message object to let the message object be constructed. It returns this message object to the Message Dispatcher, or throws an exception if the message could not be constructed or if the type does not exist.

The Message Factory encapsulates the message type table. Changes to the message types therefore only affect the Message Factory.

Code Samples

```
public class MessageFactory {

    public Message create(ByteBuffer data)
    throws MessageNotUnderstoodRuntimeException {

        assert data != null;
        assert data.hasRemaining();

        byte type = data.get(data.position());

        switch (type) {
            case ClientSuperAlive.TYPE:
                return new ClientSuperAlive(data);
            case ClientSuperJoinRequest.TYPE:
                return new ClientSuperJoinRequest(data);
```

```
            // ...

            default:
                throw new
                   MessageNotUnderstoodRuntimeException("...");
        }

    }

}
```

Resulting Context

The Message Factory is usually called as the first command of the Message Dispatcher, which receives the raw bytes from the network layer. Once the message object is created, its integrity might need to be verified using Message Verifier.

Rationale

Using the Message Factory results in a clear separation of concerns: The message objects remain responsible for serializing and deserializing themselves, while the Message Dispatcher does not need to know anything about how or which message to construct. This logic (the message type table) is encapsulated in the Message Factory, which is the only place that needs to be changed when new message objects are added or message types are changed.

Message Factory resolves all forces stated:

- The Message Factory converts the raw bytes into message objects, so that they can be further processed conveniently.
- The Message Dispatcher does not need to know how to create message objects. Instead, it simply calls the Message Factory.
- When adding a new message type, only the Message Factory needs to be adapted. This makes it very easy to add new message objects.
- The message objects can encapsulate the logic of how to marshall and demarshall themselves. The Message Factory can simply call the appropriate method.
- When changing the type of a message, it only needs to be reflected in the encapsulated type table of the Message Factory.

The solution has the following favorable qualities:

- Encapsulation: The message creation is completely encapsulated in the Message Factory, so that no other object needs to know how to create message objects.
- Extendibility: Adding new message types is easy, because only the Message Factory needs to be adapted.
- High cohesion: Using a Message Factory leads to a highly cohesive architecture. The Message Dispatcher does not need to know how to create message objects. The Message Factory on the other hand, needs to know which message object to create by reading off the type information from the raw bytes. The message creation, however, can still be encapsulated in the message object itself.
- Abstraction: The Message Factory is a good abstraction for creating message objects.
- Understandability: It is very easy to understand where messages are created when using a Message Factory.
- Clarity: Using a Message Factory leads to a clear design, because the different functions of message processing are completely separated.

Known Uses

Message Factories are used in almost all messaging systems. If high-level means provided by the programming language are used, then the underlying Message Factory is simply hidden.

In overlay networks, where messages are often marshalled and unmarshalled proprietarily, Message Factories are often used.

Type

Existing pattern (E): Factory.

References

Message Factory is a common pattern in messaging systems, where raw bytes need to be transformed into message objects. It can be regarded as a variant of the Factory pattern [19].

7.2.2 Envelope Wrapper

Context

A message from one messaging system needs to be sent with another one.

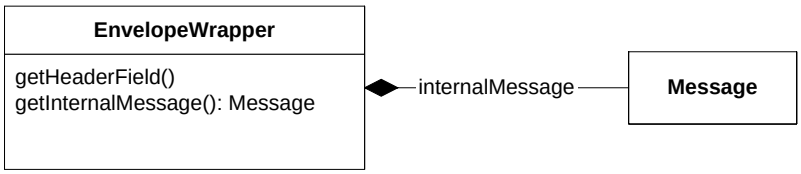
Problem

How do you send a given message with another messaging system?

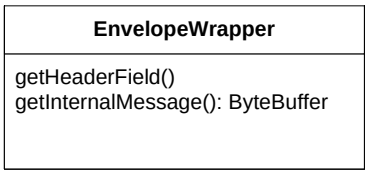
Forces

- The message to be sent belongs to a different semantic hierarchy.
- The message format in both messaging systems might be completely different.
- The message system with which the message is ultimately sent may expect some specific header fields.

Solution



or



Use an Envelope Wrapper to wrap the message to be sent in an envelope that is compliant with the message system used to send the message.

The Envelope Wrapper is a simple message from the underlying message system, so that its header fields fully comply with it. As its payload, it takes the message to be sent, which it does not need to understand itself. When the Envelope Wrapper arrives at its target, the payload is simply unwrapped and delivered

to the system for which it provided service. Because the message inside the Envelope Wrapper remains unchanged, the message is delivered in the expected format as if it was sent directly.

Resulting Context

Basic Message is an implementation of Envelope Wrapper. Routed Message is a slight modification of the same general idea.

Rationale

The solution is elegant and very simple, which makes it the preferred pattern in most messaging systems. It resolves all forces stated:

- The message that is wrapped in the envelope can belong to a different messaging system.
- The two messaging systems can have completely different formats. The Envelope Wrapper is understood by the transporting messaging system. As soon as it arrives at its target, the contained message is unwrapped and is understood by the target messaging system.
- The expected header fields are properly included in the Envelope Wrapper.

The solution has the following favorable qualities:

- Flexibility: Envelope Wrapper is a very flexible concept, because it allows any kind of message to be sent with another messaging system.
- Simplicity: Applying Envelope Wrapper is very simple; no changes are necessary to the messaging system.
- Clarity: Envelope Wrapper is a very clear and visual concept, because a message is simply contained in another one, but not converted.
- Understandability: It is very easy to understand how a message is transported and when another messaging system is used.

Known Uses

Envelope Wrapper is a base pattern which is not only used in object-oriented software systems. Its idea is applied generally in similar situations. A well-known example is TCP / IP, where TCP messages are simply wrapped in IP packets. At the receiving end, the contained TCP message is delivered up to the TCP layer of

the Internet stack. Another famous example of Envelope Wrapper is SOAP, where messages are typically send as the payload of a SOAP envelope.

In overlay networks, a variant of Envelope Wrapper is the preferred way to implemented Routed Message. FreePastry and Tapestry are just two examples where this pattern is applied.

In [32], application messages are wrapped in overlay message headers, which represent an Envelope Wrapper.

Type

Existing pattern (E): Envelope Wrapper.

References

Envelope Wrapper is well-known. A detailed description can for instance be found in 'Enterprise Integration Patterns' [35], which describes messaging patterns in general.

7.2.3 Basic Message

Context

An application uses the overlay network to route a message to a given key (e.g. a query message to localize an item in the network). Additionally, when using Extended Overlay Facade, the application also sends direct messages using the overlay network (e.g. a message to start data transfer). Altogether, the messages sent by the application have a totally different semantic from the other messages sent by the overlay network itself (Control Messages). When an application message arrives at its target node, the Message Dispatcher needs to distinguish it from Control Messages and needs to deliver it up to the application, using Application Delivery.

Problem

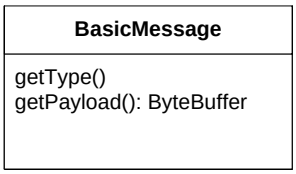
How do you separate messages sent by the application from other messages sent by the overlay network?

Forces

- The message sent by the application cannot be interpreted by the overlay network, because it may have a different format and has a different semantic.

- The message needs to be recognized as an application message and needs to be delivered up at the receiving node.
- At least one bit of information is needed to distinguish it from other messages in the overlay network.
- When using Extended Overlay Facade, not all messages sent by the application need to be routed.
- Conversely, not only messages sent by the application might need to be routed, but Control Messages can be routed as well (e.g. join messages).

Solution



Use Basic Messages to wrap messages sent by the application in a simple Envelope Wrapper. A Basic Message can be as simple as only containing the payload. However, it allows the Message Dispatcher to distinguish it from Control Messages sent by the overlay network.

When marshalling, one additional bit (e.g. the type) is sufficient to distinguish it from Control Messages. At the receiving end, the Message Factory creates a Basic Message object with the uninterpreted raw bytes as its payload, which can be delivered up to the application using Application Delivery.

Resulting Context

Basic Messages often need to be routed, in which case they should be the content of Routed Messages. Once the Basic Message arrives at its target, its payload needs to be delivered up to the application built on top. Thus, the payload will be delivered using Application Delivery.

Rationale

One bit is needed to distinguish Basic Messages from Control Messages. Therefore, using a simple Envelope Wrapper is a straightforward and elegant solution.

However, this could also be achieved with a flat message hierarchy. Basic Message serves also as a way to structure the message hierarchy nicely into application and Control Messages.

Basic Message resolves all forces stated.

- When wrapped in a Basic Message, the overlay network does not need to interpret the application message.
- However, because it is wrapped in a Basic Message, the overlay network knows that it needs to be delivered up to the application eventually.
- The information to distinguish it from other messages is encoded in the type of the Basic Message.
- The Basic Message itself is not always routed towards its target. Instead, if it needs to be routed, it can be included in a Routed Message.
- By making the concept of routable messages orthogonal to Basic Messages (see Routed Message), also Control Messages can be routed.

The solution has the following favorable qualities:

- Clarity: Using Basic Message results in a very clear structure, separating application messages from other messages in the overlay network.
- Understandability: The concept of Basic Message makes it very easy to understand which messages are sent by the application, and which are sent by the overlay network itself.
- Flexibility: Any kind of application message can be sent in a Basic Message, without any changes to the overlay network.

Known Uses

Some overlay networks do not use Extended Overlay Facade, so that all messages sent by the application can be wrapped in Routed Messages. However, in order to make that sufficient for the Message Dispatcher to know whether to deliver it to the application or not, application messages must be the only ones that may be wrapped in Routed Messages. This is true for some of the same overlay networks as well, so that in these cases, a separate Basic Message is not needed.

In all other overlay networks, some other mechanism needs to be applied. Unfortunately, there is often only one message hierarchy in the whole application.

Although this obviously works, it does not lead to a nice, readable, extensible and maintainable design, because control messages strongly differ from basic messages in their semantics. This is the reason why I have written Basic Message as a 'pattern' standing on its own.

In HyperCast [33], the overlay message header is a concrete example of this pattern [32].

Type

Existing pattern (E): Envelope Wrapper.

References

Basic Message is simply an instance of an Envelope Wrapper. In [32], Basic Messages are called *application messages*.

7.2.4 Control Message

Context

You are using Basic Message to distinguish application messages from control messages. There are a lot of control messages in your system, and all of them might share some properties and behavior.

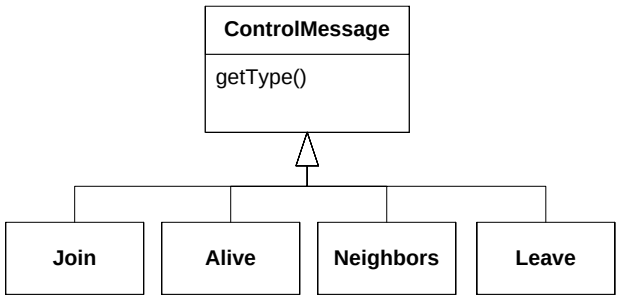
Problem

How do you structure control messages?

Forces

- All control messages might share some properties and behavior.
- Lots of different control messages exist in the overlay network.
- All control messages need to be distinguished from Basic Messages.

Solution



Use Control Message as a common super type of all control messages. This structures the message hierarchy clearly into Basic Messages and Control Messages. Additionally, common properties and behavior can be defined in the Control Message super type.

Control Message is the base class of all control messages, which inherit its properties and behavior. They extend their base class by the specific properties (header fields) they need and implement their own marshalling and demarshalling algorithm (note that it can be very helpful to generate this code automatically using a reflection-based pre-processor).

Resulting Context

Control Messages are constructed in the overlay network and sent away using the underlying network layer. When they arrive, they need to be reconstructed in coordination with the Message Factory.

Some Control Messages (e.g. join messages) might need to be routed, so they can be contained in Routed Messages as well.

Rationale

Control Message is the obvious counterpart to Basic Messages. It leads to a clearly structured message hierarchy. Using Control Message as the common super type when properties or behavior needs to be shared corresponds to standard object-oriented modeling.

Control Message resolves all forces stated:

- By having a common super type, all Control Messages can share some behavior and properties easily.

- All different control messages are cleary structured in the message hierarchy when using Control Message.
- Therefore, they are also clearly separated from Basic Messages sent by the application.

The solution has the following favorable qualities:

- Clarity: Structuring all messages of the overlay network under the common super type Control Message leads to a clear structure of the message hiearchy.
- Understandability: It is very easy to read and understand which messages are protocol messages, and which messages contain data of the application built on top.
- Extendibility: Using implementation inheritance, all Control Messages can be extended easily by simply adding or modifying fields in the Control Message super type.
- High cohesion: Because of the common super type, all common information about protocol messages are grouped together.

Known Uses

In overlay networks, Control Message is mostly applied when properties or behavior need to be shared. As a mere structuring pattern to separate Control Messages from Basic Messages, it is not used as often. However, a Control Message super type can for instance be found in P-Grid, HyperCast, Tapestry, and Azureus.

Type

Standard design solution (S).

References

In [32], Control Messages are called *protocol messages*.

7.2.5 Routed Message

Context

Routing a message to a given key is the task of the overlay network. Each node on the routing path needs to check whether it is the target of this message, or whether it needs to send it away to the next node. This logic is implemented using a Router.

However, it implies that these messages contain specific header fields, such as at least the target key of the message. If you are using Basic Message, this would be a place to include those header fields. However, it is often not true that only Basic Messages need to be routed, but Control Messages might need to be routed as well (e.g. join messages). Additionally, if you are using Extended Overlay Facade, not all Basic Messages need to be routed. At intermediate nodes, all routable messages need to be treated equally. Only at their target node, different actions need to be performed.

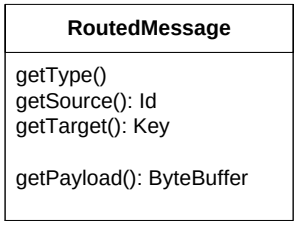
Problem

How do you make specific messages routable?

Forces

- Whether a message is routable or not is orthogonal to other message characteristics (e.g. Basic Messages and Control Messages both might be routable).
- All routable messages share some header fields, such as at least the target key.
- Code duplication and dependence on the message type are not favorable.
- Intermediate nodes should need to know what to do with the message no matter its content or specific type.

Solution



Use a Routed Message which wraps the message to be routed and adds the necessary header fields that are important for the routing algorithm.

Routed Message is a variant of an Envelope Wrapper. As its payload, it can take any message. Therefore, Basic Messages and Control Messages can both be made routable by simply creating a Routed Message containing it. Intermediate

nodes only read the fields from the Routed Messages that they need in the Router. If a message is at its target node, its content can be dispatched by the Message Dispatcher (see Router for implementation details).

Routed Message is also a variant of the Composite pattern [19], because it takes as payload any object of the common super type. However, this should not be extended much further, because it does not often make sense that Routed Messages can contain other instances of Routed Messages. This could be corrected by letting Basic Message and Control Message both extend a message class B, which extends message class A. Then, Routed Message could itself extend from A and take as payload any message of type B.

Note that if your programming language does support multiple implementation inheritance, the Routed Message could also be a super type of all those messages that need to be routable. In case your programming language only supports single inheritance, than the same effect can still be achieved if you are using automatic code generation for the message objects.

Resulting Context

When a Routed Message arrives, it needs to be checked whether it is at its target or needs to be sent to the next node on the routing path. This is taken care of in the Router pattern, which expects Routed Message objects as input. Routed Messages can contain both, Basic Messages or Control Messages, because often not only application messages need to be routed, but 'join' messages for instance as well.

Rationale

The solution takes into account that several messages can be routable. Thus, it would not be flexible to add the header fields needed for the routing algorithm only to a specific message type. Furthermore, the abstraction is very nice, because the actual message that is routed is simply the payload of an Envelope Wrapper message.

Routed Message resolves all forces stated:

- By using Routed Message, both, Basic and Control Messages, can be routed.
- The common header fields are given by the Routed Message.
- The Router and Message Dispatcher does not need to check for different types of messages that are routable. Instead, it can treat all Routed Messages the same way.

- Also intermediate nodes can treat all Routed Messages the same way, no matter the content.

The solution has the following favorable qualities:

- Flexibility: Using Routed Message, it is trivial to make any kind of message routable. The mechanism is thus very flexible.
- Encapsulation: The Routed Message encapsulates all the header fields necessary for the routing logic.
- Clarity: Using Routed Message leads to a very clear structure. If a message is routed or not is orthogonal to the message hierarchy.
- Abstraction: Routed Message is a very nice abstraction for all messages that are routed.
- Understandability: It is easy to see whether a message is routed or not, and it is easy to see how an intermediate node treats a Routed Message.
- High cohesion: Routed Message leads to high cohesion in the message space; the concept of routed messages is encapsulated in the Routed Message, while the specific message that is routed is encapsulated by itself.
- Avoids code duplication: Because the concept is factored out of individual messages, no code duplication needs to take place.

Type

Adaptation of an existing pattern (A): Envelope Wrapper.

Known Uses

Almost all overlay networks use a variant of Routed Message as a basic concept (e.g. FreePastry, Tapestry). FreePastry combines Routed Message with a variant of Autonomous Message.

7.2.6 Specialized Message Type

Context

In real systems, the overlay network often consists of different types of nodes (e.g. super, storage, and client nodes). All these different nodes are connected together, so that lots of messages need to be sent in total. The messages sent

between different node types are often very similar in their intent and behavior. One example of such a universal message type is an 'alive' message that can usually be sent between any two nodes to inform the receiver that the sending node is still active. However, using the same messages between any pair of nodes can make it hard to test, detect faulty conditions and monitor the network. Additionally, the content of universal messages is sometimes misused to convey different kinds of information, thus strong types cannot be used.

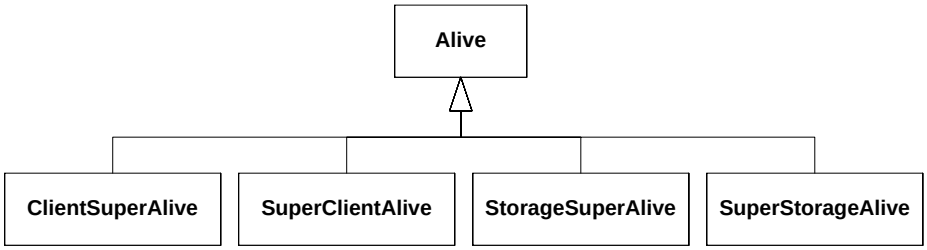
Problem

How can you improve testability on the messages and profit from static type-safety to render some faulty network conditions impossible?

Forces

- For each pair of nodes, similar messages need to be sent. Using the same message for each pair of nodes is therefore often possible.
- For monitoring and testing, looking at the sender and receiver of a message can yield some information.
- Sometimes, however, the information given by the sender and receiver is not enough to unambiguously deduce which node has sent which message. This is for instance the case if several nodes can be hosted on one computer, but the sender and receiver addresses only correspond to the Internet address of the computer.
- Using universal messages, testing and debugging the protocol is much harder because messages cannot be distinguished early, or by simply looking at their types.
- Messages sometimes convey information such as Node Handles. When using universal messages, these fields cannot be strongly typed, thus for instance Typed Node Handle cannot be applied in the messages.
- Creating new message types takes some effort. They need to be added to the Message Factory, and if the marshalling and demarshalling algorithm must be written manually, it is tedious and error-prone.

Solution



Use Specialized Message Types, which allows monitoring exactly which messages two nodes exchange. It also helps for debugging and allows each message to be adapted to the specific needs of each pair of node types, thus including strong types for the fields. Wrong assignments can therefore already be checked by the compiler.

Specialized Message Type splits the universal 'alive' message into one for each different pair of node types. It therefore distinguishes between 'super-client-alive', 'client-super-alive', 'super-storage-alive', etc. This makes it easier to monitor the network and to debug. Additionally, each message can now contain strongly typed fields, which renders faulty assignments impossible, which were hard to detect otherwise. Static type-safety can further be improved if message types can only be sent off if their target address corresponds to the correct node type. A 'super-client-alive' message may only be sent off, if the target Node Handle is a Typed Node Handle of type client. Together with Source Sink Marker, this can be easily checked. Furthermore, the statistics provided by a Traffic Monitor can be much more accurate when using Specialized Message Types.

The drawback of this pattern is that creating new message types takes some efforts, especially if a proprietary marshalling and demarshalling code must be written. To overcome this, it can be very helpful to write an automatic reflection-based code generator.

When using Specialized Message Types, there are many more messages in the message hierarchy. Therefore, a good naming convention should be applied. Using explicit names that contain the sending node and the receiving node (e.g. SuperClientAlive) can be a good choice, because it makes it very clear to the reader of the code. Additionally, Source Sink Marker can be used to state this information more explicit and to even check it in the code.

Resulting Context

When using Specialized Message Types, the Message Factory needs to be updated for each type of message. The Message Dispatcher needs to dispatch each message to the object responsible for processing it. If you are using Message Handler in combination with Local Node For Each Type, then all messages sent to a given node type will be processed by the same Local Node. Therefore, it would be nice to detect the sink node type of a message automatically. This problem is addressed in Source Sink Marker.

Rationale

Using Specialized Message Types corresponds to the object-oriented approach of always using the most specific type possible. This allows specifying the system as accurate as possible, avoiding any ambiguities. Because the specific messages can then contain strong types, some mistakes can already be detected by the compiler. Additionally, using Specialized Message Types improves testability and allows to monitor the network much more accurately.

Specialized Message Type resolves most forces stated:

- Although it is possible to send the same message between different pairs of nodes, Specialized Message Type has a number of advantages, such as improved testability and static type-safety, so that it can be better in some cases.
- While the sender and receiver can yield some information, it is not always possible to deduce the type of the nodes unambiguously. Specialized Message Type is a simple solution for this problem.
- In the case of virtual nodes, sender and receiver node type can be deduced when using Specialized Message Type.
- Using Specialized Message Type, it is easy to test, debug and monitor the overlay network, because only the interesting Specialized Message Type needs to be tracked.
- Specialized Message Type also allows to contain type-safe data. Instead of Node Handles, Typed Node Handles can be used for instance.
- However, adding new message types for each pair of message takes some effort and bloats the message hierarchy.

The solution has the following favorable qualities:

- Type-safety: Not only can it be checked whether a certain node type can receive a certain message type, but the messages itself can also contain type-safe information, such as Typed Node Handle instead of Node Handle.
- Clarity: Introducing a new type of message for each pair makes it very explicit and therefore clear which messages are exchanged by which node types.
- Understandability: The explicit structure makes it very easy to understand the purpose of each message.
- Testability: With Specialized Message Types, it is much easier to test and debug the network, because the exchange of a message between a certain pair of nodes can be tracked individually.

Unfortunately, the solution misses the following qualities:

- Flexibility: Using general message types is much more flexible, because no adaptions are needed when adding new node types.
- Effortless: Adding a new message type for each pair of node takes some effort.
- Simplicity: The message hierarchy is more complex than if there was only one general message type. However, while at first glance it is more complex, reading the source code becomes much simpler.

However, taking the extra effort needed can be justified by the benefits of a clear, explicit structure that improves testability.

Known Uses

Unfortunately, Specialized Message Type is not used that often, but universal message types are sent between each pair of node. In a lot of academic projects, however, there is often only one type of node, so that Specialized Message Type cannot even be applied.

Type

Standard design solution (S).

7.2.7 Source Sink Marker

Context

If you are using Specialized Message Types, each message should be sent and received by exactly one type of node. Explicit names for the message classes, containing the sender (source) and receiver (sink) node type, can help the reader of the code. However, the Message Dispatcher still needs to know each message type and reasoning about the source and sink node type in the code is not possible (e.g. in the Traffic Monitor).

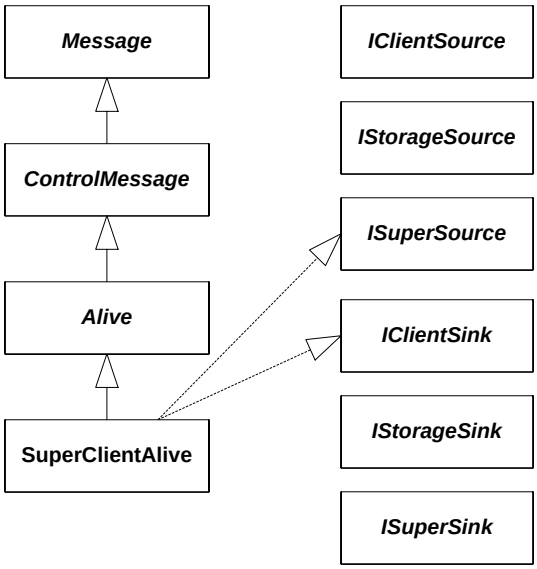
Problem

How can you make the source and sink node type of a message explicit, so that it can be checked in the code?

Forces

- Using long class names to indicate the source and sink of a message is very helpful for the programmer reading the source code, but is not sufficient for the dispatching mechanism to distinguish the types as well.
- Using a separate table which lists the source and sink of each message is a source for inconsistency and duplication. Furthermore, it does not improve readability when browsing through the code.
- Using reflection to reason about the source and sink node type from its class name is not efficient and safe.

Solution



Use Source Sink Marker to mark the source and sink node type of each message. Let each message simply implement this (empty) Marker Interface, so that the node types can be checked programmatically.

Simply create a source and a sink Marker Interface for each type (e.g. *IClientNodeSource*, *IStorageNodeSource*, *ISuperNodeSource*, *IClientNodeSink*, *IStorageNodeSink*, *ISuperNodeSink*), and let each message implement the corresponding interface. If you are using Local Node For Each Type, then the dispatching mechanism becomes very simple; only the sink type of a message has to be checked (e.g. in Java using *instanceof*) in order to dispatch it to the appropriate local node object.

Because Source Sink Marker introduces explicit types for the messages, some mistakes can be detected already at compile-time. Other mistakes can be detected by using assertions at run-time. It can for instance be checked that messages for a client node can only be sent off if the target node is of type client (using Typed Node Handle).

If the programming environment easily allows seeing the interface of an object, then it is also more readable for the programmer. At least when a programmer looks at the class, it becomes clear who sends and who receives this type of message. To improve readability further, it can also be a good choice to use explicit names for the message types, containing the source and sink node type (e.g. *SuperClientAlive*).

Code Samples

```
public class MessageDispatcher {

    public void dispatch(InternetAddress sender, ByteBuffer data) {
        assert sender != null;
        assert data != null;

        Message message = messageFactory.create(data);

        // ...

        if (message instanceof RoutedMessage) {
            // ...
        } else if (message instanceof BasicMessage) {
            // ...
        } else if (message instanceof ControlMessage) {
            if (message instanceof IClientNodeSink) {
                ClientNode clientNode = machine.getClientNode();

                if (clientNode != null) {
                    clientNode.handle(message);
                } else {
                    throw new MessageHandlingException("...");
                }
            } else if (message instanceof IStorageNodeSink) {
                Collection<StorageNode> storageNodes = machine.getStorageNodes();

                if (!storageNodes.isEmpty()) {
                    for (StorageNode storageNode : storageNodes) {
                        storageNode.handle(message);
                    }
                } else {
                    throw new MessageHandlingException("...");
                }
            } else if (message instanceof ISuperNodeSink) {
                SuperNode superNode = machine.getSuperNode();

                if (superNode != null) {
                    superNode.handle(message);
                } else {
```

```
                    throw new MessageHandlingException("...");
                }
            }
        }
    }

}
```

Resulting Context

When using Local Node For Each Type and Message Dispatcher, each Local Node needs to register itself at the implicit Message Dispatcher for the messages it is interested in. Likewise, if an explicit Message Dispatcher is used, it needs to dispatch the messages off to the Local Node depending on their type. Once Source Sink Marker is applied, this becomes very simple, because now only the sink type needs to be checked.

Rationale

Source Sink Marker introduces strong types and therefore makes the system more explicit and safe. Using Source Sink Marker, the source and sink node type of a message can be checked programmatically, which can help at different places (e.g. Message Dispatcher, Traffic Monitor).

It resolves all forces stated:

- Using Source Sink Marker allows the Message Dispatcher and other places in the code to check the source and sink node type of a message programmatically.
- Source Sink Marker states the type of the source and sink of a message at the right place, the message itself. This avoids inconsistencies and improves readability.
- Checking the type of an object is much more efficient and safe than using reflection.

The solution has the following favorable qualities:

- Clarity: Source Sink Marker makes it explicit and easy to check the source and sink node type a message programmatically.

- Understandability: When reading the code, this extra information helps to understand it quicker.
- Simplicity: Using Source Sink Marker, the Message Dispatcher becomes much simpler, because messages can simply be dispatched off according to their sink node type.
- Effortless: Using Source Sink Marker gains some of the effort that was spent to create each different Specialized Message Type, because in the Message Dispatcher, most often only the sink type of a message needs to be checked to decide which Message Handler to dispatch it to.

Known Uses

Marker Interfaces are used a lot in different software systems. Prominent examples are for instance the Serializable and Remote interface in the Java programming language.

For the purpose of marking the source and sink of messages, I could not find this pattern be applied in overlay networks so far.

Type

Existing pattern (E): Marker.

References

Source Sink Marker is a concrete instance of the Marker Interface pattern, which is a well-known base pattern. A description of the Marker Interface pattern can for instance be found in [64].

7.3 Message Handling

Messages arriving at a node need to be processed. They either need to be delivered up to the application, sent further to the next node on the routing path, or handled by the overlay network itself to maintain the routing topology. This section introduces patterns describing how messages are received and processed. Similar to the patterns from the last section, these patterns are known from distributed systems and are simply adapted in here for the special requirements of overlay networks.

7.3.1 Message Dispatcher

Context

When a message arrives, the network layer delivers the raw bytes that make up a logical message up to the overlay network layer. You are using Message Factory to transform the raw bytes into a message object. Depending on the actual type of the message, different actions need to be performed in the overlay network.

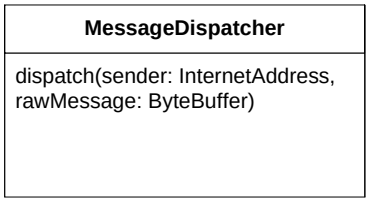
Problem

How do you process the different messages?

Forces

- Processing all messages at one single place is not favorable.
- Despite the large number of messages, it should be easily readable how and where each message is processed. The execution flow should be very clear.
- Adding new message types should be very simple.
- Some messages need to be delivered to the application.
- Messages might be corrupted. However, they should not result in an unstable state of the overlay network.

Solution



Use a Message Dispatcher, which receives the raw bytes from the network layer, creates the corresponding message object using the Message Factory, and dispatches it to the actual object responsible for processing it.

Usually, the Message Dispatcher needs to conform to an interface defined by the network layer (similar to Application Delivery for the overlay network layer).

Whenever a message arrives, the network layer will call this method to deliver the raw bytes to the overlay network layer.

The actual implementation of the Message Dispatcher can look very differently. It can either be an explicit or an implicit Message Dispatcher. In the explicit Message Dispatcher, it is explicitly stated which message object to dispatch to which object (e.g. using a *switch* statement on the message type). In an implicit Message Dispatcher, Message Handlers register themselves with it, so that it does not know whom to dispatch a message object statically, but dispatches it to whoever registered it for that message type (e.g. using a list that stores the type together with the Message Handler object).

Both implementations are feasible and have their own advantageous. An explicit Message Dispatcher has the advantage that the execution flow can be followed explicitly by reading the code of the dispatcher. In the implicit dispatcher, different objects register themselves with the dispatcher, so readability is hindered by the need to read different places making up the dispatcher logic. On the other hand, explicit Message Dispatchers are more rigid, whereas implicit Message Dispatchers can be extended with new message objects easily. However, this argument is not as strong if one takes into account that both solutions require about one or two lines of code that need to be added at the maximum for each message type, and using for instance Source Sink Marker might make it unnecessary in both cases. In general, implicit Message Handlers are typically the preferred method when Message Handlers need to be registered that are not known at compile time. This if often the case in frameworks or libraries, where new handlers need to be registered without changing the source code of the dispatcher (e.g. event listeners in GUI frameworks). In this case, however, the overlay network is usually under full control of the programmer and no handlers need to be added at runtime (compared to the discussion in Application Delivery, where a more flexible mechanism is needed).

Because the Message Dispatcher does have a special relation with Application Delivery and Router, the explicit dispatcher might make this clearer. An implicit Message Dispatcher is only possible if all objects responsible for processing messages conform to a common interface (the Message Handler). This would only be possible by implementing this for Application Delivery and Router as well, which is certainly possible, but not necessarily favorable. However, using a Message Handler for all other objects is strongly recommended, even in the case of an explicit Message Dispatcher.

To guarantee stability even in the case of corrupted messages (wrong content), the Message Dispatcher needs to apply meaningful exception handling mechanisms and makes use of the Message Verifier.

Code Samples

```
public class MessageDispatcher {

    public void dispatch(InternetAddress sender, ByteBuffer data) {
        assert sender != null;
        assert data != null;

        Message message = messageFactory.create(data);

        // ... see Message Verifier, Traffic Monitor

        // dispatch off messages
    }

}
```

Resulting Context

The Message Dispatcher will need the Message Factory to first create the message object. Because some messages need to be checked for their integrity, Message Dispatcher can make use of a Message Verifier. The Message Dispatcher dispatches the messages off to the appropriate object, which should be a Message Handler. The Message Dispatcher needs to work closely together with the Router in case a Routed Message is not at its target, but needs to be routed to the next node. If a Routed Message containing a Basic Message (issued by the application built on top) has arrived at its target node, the Message Dispatcher needs to deliver the message up to the application, using Application Delivery. If you are using an explicit dispatcher, then Source Sink Marker can be helpful to make the dispatcher be lean and elegant.

If there is only one node per computer, or if each node listens on a different port (demultiplexing at a lower level), then the Message Dispatcher can be combined with Local Node.

Rationale

The Message Dispatcher is only responsible for dispatching the messages to the appropriate objects. These Message Handlers are then responsible for processing the message (separation of concerns). Adding new message types is very simple, because the object processing a new message type only needs to be registered at the Message Dispatcher.

The Message Dispatcher resolves all forces stated:

- Instead of processing the messages directly when they enter the overlay network, the Message Dispatcher dispatches them off to the appropriate object processing the message.
- Using a Message Dispatcher results in a very clear structure regarding incoming messages.
- When adding new message types, only the Message Dispatcher needs to be changed (as well as the Message Factory of course).
- The Message Dispatcher can deliver Basic Messages up to the application.
- The Messsage Dispatcher can drop corrupted messages. In cooperation with the Message Verifier, it easy to treat all messages equally if they are corrupted.

The solution has the following favorable qualities:

- High cohesion: Using a Message Dispatcher to simply dispatch off incoming messages leads to high cohesion, because the Message Dispatcher is only responsible for dispatching the messages, while the Message Handler is responsible for processing it.
- Clarity: Using a Message Dispatcher leads to a very clear design, because it is made explicit where messages enter the overlay network.
- Understandability: Using a Message Dispatcher, it is easy to see which Message Handler is responsible to process which message, and thus the incoming message objects can easily be tracked and understood.
- Size: When the Message Dispatcher is only responsible to dispatch the message objects, it remains very small and understandable.

Known Uses

Most distributed systems use Message Dispatcher to dispatch off messages. Also in overlay networks, this pattern is dominantly used for this purpose. FreePastry combines Message Dispatcher with Local Node. Tapestry and Bamboo combine Message Dispatcher with Router. In some projects, the Message Dispatcher does not really dispatch the messages off, but processes them by itself, thus leading to long and obfuscated dispatching logic.

Alternative Patterns

Autonomous Message is closely related to a Message Dispatcher, because it provides another solution to the same problem. If different messages result in the same action, Message Dispatcher might be the better alternative.

Type

Existing pattern (E): Observer.

References

This pattern is well-known from lots of distributed systems, and it has been documented several times. It can be regarded as a variant of the Observer pattern, at least in the implicit case. The responsibility of the Message Dispatcher in overlay networks is a little bit extended compared to other systems, because Routed Messages require special processing. This, however, depends on the actual implementation used (see Router).

7.3.2 Message Handler

Context

You are using Message Dispatcher to dispatch off messages to the objects responsible for processing them. If you are using an implicit Message Dispatcher, all objects responsible for processing a message object need to conform to a common interface. If you are using an explicit Message Dispatcher, there is no such obligation. In either case, the Message Dispatcher should be as simple as possible.

Problem

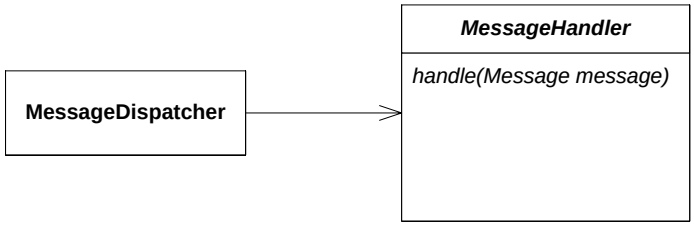
How can you make the Message Dispatcher be as simple and small as possible?

Forces

- The Message Dispatcher should only need to dispatch off messages to the appropriate objects. It should not process the messages directly.
- A message should be processed as late as possible, meaning that it should be processed as close to the corresponding object as possible.
- The execution flow when messages arrive should be very clear. It should be very easy to see how a message is processed and who is responsible for it.

- Processing message objects directly can be more efficient.

Solution



Use a Message Handler interface which simply provides a method *handle(Message message)*. All objects that are responsible for processing messages need to implement this interface. Then, the Message Dispatcher can call this method to dispatch off messages and is prevented from processing them by itself.

For implicit Message Dispatchers, it is mandatory that all objects implement the Message Handler interface. But even for explicit Message Dispatchers, it might be beneficial because it results in a clear and standardized design. Because the *handle()* method takes message objects as arguments, it forces the Message Dispatcher to dispatch off the messages, rather than processing part of them by itself, leading to long and obfuscating code.

The object responsible for processing messages should implement the Message Handler interface directly. If there are lots of messages for which it is responsible, however, it might be worthwhile to refactor the internal dispatching out into an inner class (or even a separate class). The implementation of the *handle()* method would then simply call another handle method on an instance of its internal message handler class.

Resulting Context

The Message Dispatcher dispatches messages off to Message Handler objects. Local Nodes are a perfect example of where to apply Message Handlers. An implicit Message Dispatcher can simply store a list of message types together with their Message Handler object, and then look up this list to know whom to dispatch the message to. Application Delivery and Router are a special case of Message Handlers. Of course, also these objects could implement a Message Handler. Because of their special role, however, this can be implicit so that the methods *deliver()* and *route()* are called explicitly. This implies that the implicit dispatching can

only be applied for other Message Handlers (handling especially Control Messages) in the overlay network. It also makes a stronger point on why explicit Message Dispatchers can be preferred in overlay networks.

Rationale

Of course, this solution is very simple and straightforward. It is a key component of the Observer pattern, or event listeners in general, where *Observable* and *Listener* interfaces fulfill the role of Message Handlers. The reason why this solution is written as a 'pattern' on its own is to make another point. Because using Message Handlers in explicit Message Dispatchers is not an obligation, it often leads to long and obfuscating code in the dispatcher. The dispatcher would typically unwrap some of the content contained in the message and would call different methods on helper objects (such as Local Nodes) in order to process it. If the logic is only small, it would sometimes do the whole processing by itself and would even send it off using the underlying network layer. By stating this as its own pattern, it gives reasons to use Message Handlers explicitly for every object processing messages, resulting in small and easy maintainable Message Dispatchers.

Delaying the message processing by dispatching it off to Message Handlers in every situation actually trades off efficiency for clarity. However, it can be argued whether this really results in significant efficiency losses due to method calls.

Message Handler resolves most forces stated:

- Using Message Handlers makes it explicit to not process messages in the Message Dispatcher itself, resulting in a lean Message Dispatcher and a clear structure.
- Using Message Handlers results in a highly cohesive architecture, where messages are processed as close to the appropriate object as possible.
- Using a common interface makes it clear and readable where messages arrive and how they are processed.
- As already noted, delegating the messages off to Message Handlers is slightly less efficient than processing them directly.

The solution has the following favorable qualities:

- High cohesion: When using Message Handlers, it is made very explicit that the Message Dispatcher may not process the message objects itself, leading to high cohesion.

- Size: Because the Message Dispatcher does not process messages itself, it remains very small.
- Clarity: It is made explicit where and how message objects are processed, leading to a clear structure.
- Understandability: It is easy to track message objects and understand how and by which object they are processed.
- Flexibility: It is very easy to add new Message Handlers.
- Encapsulation: The processing logic is completely encapsulated by the Message Handler.
- Abstraction: Using Message Handlers gives a nice abstraction in the overlay network, because the message flow and processing is made more explicit.

Known Uses

Message Handler is used whenever an implicit Message Dispatcher is applied. This is the case for instance in FreePastry, where Message Handlers implement the *MessageReceiver* interface. Unfortunately, where explicit Message Dispatchers are used, Message Handler is rarely applied.

Type

Existing pattern (E): Observer.

References

Message Handler corresponds to the well-known *Observer* interface in the Observer pattern [19], and to event listeners in general.

7.3.3 Autonomous Message

Context

A Message Dispatcher dispatches off messages to the appropriate object processing it. This implies two things. First, the conditional logic in the Message Dispatcher needs to be maintained when the message hierarchy changes, and second, Message Handlers are processing message objects, rather than message objects processing themselves.

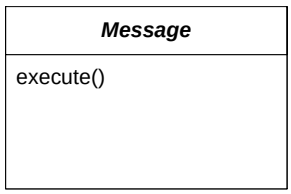
Problem

How could you let messages process themselves, making the Message Dispatcher and Message Handlers become redundant?

Forces

- Maintaining the conditional dispatching logic in the Message Dispatcher takes some effort.
- When using Message Handlers, processing logic is not encapsulated in the message object itself.
- Processing message objects by Message Handlers is 'function-oriented', not truly object-oriented.
- Letting messages process themselves makes it harder to read all message processing logic, because it is encapsulated in different message objects.
- Simply letting messages dispatch themselves to Message Handlers does not bring a lot of advantages.
- If different messages are processed similarly, the code will be very similar as well.

Solution



Use Autonomous Messages, which know how to process themselves. They are processed by calling an *execute()* method on the message object.

When a message enters the overlay network, it first needs to be transformed into a message object, using Message Factory. After that, the *execute()* method of the message object is called to let it process itself. The object which first calls the Message Factory and then executes the message object can be regarded as a very simple Message Dispatcher. However, the whole conditional logic of

which message to dispatch to which Message Handler is not necessary anymore. Additionally, Message Handlers are not needed any more either, because messages process themselves completely.

A Routed Message, for example, will interact together with the Router to check whether it is at its target node. If so, it will unwrap the internal message and let it be created and executed by the same simple Message Dispatcher. If not, it will make itself be sent to the next node on the routing path. Basic Messages will deliver themselves up to the application using Application Delivery. Control Messages perform the appropriate action in the overlay network themselves, for instance adding Node Handles to the routing table or refreshing entries in the neighbors table.

Code Samples

```
public abstract class Message {

    public void execute();

}

public class MessageDispatcher {

    public void dispatch(InternetAddress sender, ByteBuffer data) {
        assert sender != null;
        assert data != null;

        Message message = messageFactory.create(data);
        message.execute();
    }

}
```

Resulting Context

Autonomous Messages will either be Control, Basic or Routed Messages. Routed Messages need to interact with the Router. They also need to work together with Network Gateway or Node Handle Proxy to transmit themselves using the network layer. Basic Messages need to deliver themselves up using Application Delivery. Control Messages will make use of different objects from the overlay network, such

as Local Node.

Rationale

Autonomous Messages implement the whole message processing in a truly object-oriented manner. Message Dispatchers and Message Handlers disappear, and processing code is close to the object itself, in its own *execute()* method, rather than spread all over in the overlay network.

However, Autonomous Messages may also have some severe drawbacks. If several messages need to be processed similarly, the *execute()* method will contain very similar code. Such code duplication can only be avoided if the message objects can inherit their behavior from a common super type, which is not always possible using single implementation inheritance due to the complex message hierarchy. If the processing code would be refactored out of the message object, the solution will soon resemble the Message Handler pattern, and the only logic that is contained in the message object will be the dispatching logic. Then, however, a central Message Dispatcher might provide a better overview.

Furthermore, it is not clear whether the *execute()* should only be called after a message is constructed through the Message Factory, thus representing an incoming message, or whether it can also be called for messages that are created locally meant to be sent off. To overcome this, one could split each message object in two, an incoming and an outgoing message object. The incoming message object would process itself, whereas the outgoing message object would sent itself off over the network. Both, the incoming and the outgoing message object, would extend their common super type, which contains all fields and shared behavior. However, the *execute()* method would be overridden by each type. Such a design would not only bloat the message hierarchy, but also lead to code duplication, at least in the outgoing message object's *execute()* method.

However, Autonomous Messages can be a good solution in some occasions, because the processing logic is encapsulated in the message object itself. There is a similar pattern known in the literature, the Command Message, a variant of the Command pattern, which has been successfully used in different systems. However, it is important to note that Autonomous Messages differ significantly from Command Messages (see Reference).

Because of the listed drawbacks, it is arguable whether Autonomous Message should be used at all. At first, it is very tempting, but it later may prove to have more disadvantages than advantages. In these cases, Autonomous Message could possibly also be an 'anti-pattern' [65] in some contexts.

Autonomous Message resolves some of the stated forces:

- The dispatching logic does not need to be maintained in the Message Dispatcher, but is included in the Autonomous Message itself.
- The processing logic is encapsulated in the message object.
- Therefore, Autonomous Message corresponds to straightforward object-oriented modeling.
- Unfortunately, when using Autonmous Message, it is harder to read the processing logic for different message objects, because it is strongly encapsulated.
- Instead of dispatching themselves to Message Handlers, Autonomous Messages should only be applied if they can be executed autonomously, not depending on lots of other objects.
- If different messages need to be processed very similarly, Autonomous Message is not the best choice, because it results in code duplication.

The solution has the following favorable qualities:

- High cohesion: Autonomous Messages lead to high cohesion, if all the processing logic can be contained in the message object itself.
- Low coupling: Then, it results in very low coupling as well. However, it is rarely the case that a message can be executed very autonomously, without relying on other objects, because it most often needs to change the state of the environment at its target.
- Encapsulation: The Autonomous Message encapsulates both, data and behavior, in itself.
- Flexibility: Adding new message types results in almost no changes in the rest of the overlay network, not even in the Message Dispatcher and Message Handlers.

Known Uses

FreePastry uses a variant of Autonomous Message for its Routed Message. The *RouteMsg* object contains a method *routeMessage()* which routes a message off to the next node, given that the next node is already set up by the Router. However, this is the only place where it is used in FreePastry, and other messages are dispatched off and processed normally. However, Autonomous Message is also used in some other general peer-to-peer systems.

Alternative Patterns

Message Dispatcher together with Message Handler is an alternative to this pattern. It may be better in situations where several messages need to be processed similarly.

Type

Adaptation from an existing pattern (A): Command Message.

References

Autonomous Message closely resembles the Command pattern [19]. In the Command pattern, commands are wrapped as objects with their appropriate *execute()* method. This pattern has been extended to message objects, known in the literature under the name of Command Message (e.g. [35]), which has been successfully used in a lot of systems. Similar to Autonomous Message, Command Messages can be executed after they are received. However, there is an important difference to Autonomous Message. Command Messages contain commands in their message as content. The contained source code can be executed at the target using reflection mechanisms or scripting functionality. In Autonomous Messages, no source code is contained when sent over the network. It is simply a matter of design where the processing logic is contained, thus it does not enhance flexibility.

7.3.4 Message Verifier

Context

Peer-to-peer systems are very vulnerable to attacks because they are inherently open and exposed. One specific form of attack is by sending messages with a forged sender address, that is, messages claiming to be from a certain node, while they are in reality sent by an attacker. You want to include a simple yet effective mechanism to detect such messages. Messages need to carry cryptographic proofs (credentials, signatures) in order to verify the sender of a message. However, credentials can be quite large, so that they could bring down overall efficiency if they were included in all messages. Because message integrity is not equally important for all messages, credentials should be included in some specific messages only. For all other messages, you may want to perform some simple verification checks.

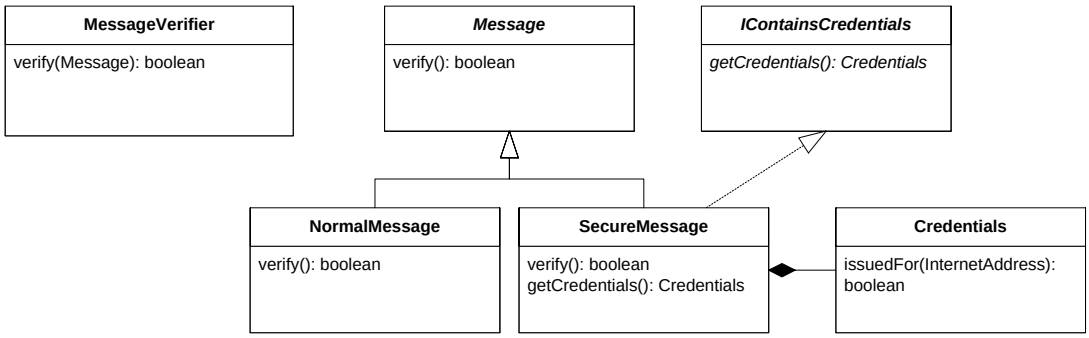
Problem

How can you include a simple verification mechanism, which can be extended for specific messages?

Forces

- Forged messages should be detected early, before they can cause damage.
- Some message should include credentials. It should be easy to add credentials to existing messages or to new ones.
- Verifying the credentials is similar for all messages containing credentials.
- The message hierarchy can be quite complex.
- Simple verification can be done by checking header fields.
- Each message may include its proper verification logic.

Solution



Use a Message Verifier, which provides a method *verify()* that checks the integrity of all messages. If credentials are included, it will verify that they are indeed issued for the claimed sender.

The Message Verifier provides a central place for doing all verification checks. The *verify()* method can first do some simple checks on the header fields or do other integrity checks. Then, it calls the *verify()* method of each message, which allows each message to implement its own verification algorithm. Furthermore, if a message contains credentials, it checks whether the credentials are indeed issued for that sender, thus verifying that the message is really sent by the claimed

sender. If the Message Verifier cannot verify a message, it can be dropped by the Message Dispatcher, so that it cannot cause any damage.

The *verify()* method of each message is provided by the common super type Message. The Message class simply implements this method by returning *true*. If a specific message type wants to implement its own verification algorithm, it can simply override this method.

Interesting checks can only be done if a message contains cryptographic proofs. This allows for instance to verify that the sender is really who he claims to be. All messages containing such credentials need to implement a common interface (e.g. *IContainsCredentials*), so that the Message Verifier knows that it needs to check the credentials. Messages implementing this interface need to implement a method similar to *getCredentials()*, which returns the credentials so that the Message Verifier can perform the checks.

Code Samples

```
public class MessageDispatcher {

    public void dispatch(InternetAddress sender, ByteBuffer data) {
        assert sender != null;
        assert data != null;

        Message message = messageFactory.create(data);

        if (!messageVerifier.verify(sender, message)) {
            // message not verified
            // perform appropriate action
            return;
        }

        // ...
    }

}

public class MessageVerifier {

    public boolean verify(InternetAddress sender, Message message) {
        assert sender != null;
```

```
        assert message != null;

        // do basic checks reflecting general policies

        if (!message.verify()) {
            return false;
        }

        if (message instanceof IContainsCredentials) {
            Credentials credentials = ((IContainsCredentials) message)
             .getCredentials();

            if (!credentials.verifyOwner(sender)) {
                return false;
            }
        }

        return true;
    }

}

public interface IContainsCredentials {

    public Credentials getCredentials();

}

public class Credentials {

    public boolean verifyOwner(InternetAddress sender) {
        assert sender != null;

        // ...
    }

}

public class Message {
```

```
    public boolean verify() {
        return true;
    }

}

public class NormalMessage extends Message {

    public boolean verify() {
        // check fields and content
    }

}

public class SecureMessage extends Message implements IContainsCredentials {

    private Credentials credentials;

    public boolean verify() {
        // check fields and content
    }

    public Credentials getCredentials() {
        return credentials;
    }

}
```

Resulting Context

Messages need to be verified when they arrive in the overlay network. After creating the message using Message Factory, the check needs to be done immediately, before dispatching or processing the message further. Thus, the Message Dispatcher calls the Message Verifier just after the message has been created. In case it is not verified, the Message Dispatcher either drops (and logs) the message, or performs any other appropriate action.

Note that some simple semantic checks can be included by always using strong types. Therefore, it is recommended to use Typed Node Handle to include in messages, which also gives reasons to use Specialized Message Types.

Rationale

In the case of single implementation inheritance, the credentials cannot be verified by a method provided by a common super class, because of the complex message hierarchy and the fact that only specific messages should include credentials. Therefore, this logic needs to be factored out into a Message Verifier. Messages that include credentials need to be recognized and need to provide a common method *getCredentials()* to get the contained credentials. This makes it very easy to include strong security for specific messages only, and adding credentials to existing or new messages is trivial. If multiple implementation inheritance can be used, then the verification method could also be provided by a common super class. Aspect-oriented programming could also be applied for this purpose.

The simple *verify()* method provided by the common super class Message can be overridden by specific messages to do some simple checks on other fields or on the content. In the simplest case, the method just returns *true*.

The Message Verifier also provides a central place to include other verification checks, reflecting common policies on an abstract level, orthogonal to individual messages (for instance to drop all messages sent by a specific address).

The Message Verifier resolves some of the forces stated:

- If the Message Verifier is called as soon as the Message Factory has created the message object, forged messages are detected early and can be dropped by the Message Dispatcher.
- Not all messages need to contain credentials, because it is an overhead and makes the message larger, consuming more bandwidth. However, it is very easy to include credentials for any kind of message that needs it.
- All messages containing credentials are verified the same way.
- The Message Verifier pattern takes into account that the message hierarchy can be very complex, and that for instance the credentials cannot always be inherited.
- Simple verification of header fields, or applying a general policy, is very easy using the Message Verifier.
- Each message can include its own verification logic.

The solution has the following favorable qualities:

- Flexibility: The Message Verifier is a very flexible mechanism. Any kind of message can be verified, and the level of security can be increased stepwise. Credentials are completely orthogonal to the rest of the message hierarchy, and every message object can implement its own verification logic.
- High cohesion: Verification logic is implemented in three parts, but access and checks are encapsulated in the Message Verifier.
- Understandability: It is easy to understand when and where verification logic is executed, and what happens in the Message Dispatcher if a message is not verified.
- Clarity: The Message Verifier is a very clear and visual structure for where verification of messages takes place.

Known Uses

FreePastry is designed to include a Message Verifier. JXTA uses credentials and certificates in an extended way.

Type

Proto-pattern (P).

7.4 Routing

Routing messages to specified keys is the main task of the overlay network. The actual routing algorithm is highly specific to the overlay network, but the solution to its software design can be reused for all overlay networks. This section introduces the pattern which is used in most overlay networks.

7.4.1 Router

Context

The task of the overlay network is to route messages to specified keys. To achieve this goal, every node receiving a message sends it to the node from its routing table with the identifier which is closest (in whatever metric used) to the key. Once the message arrives at its target node, it needs to be delivered up to the application. You are using Routed Message to identify such messages.

Problem

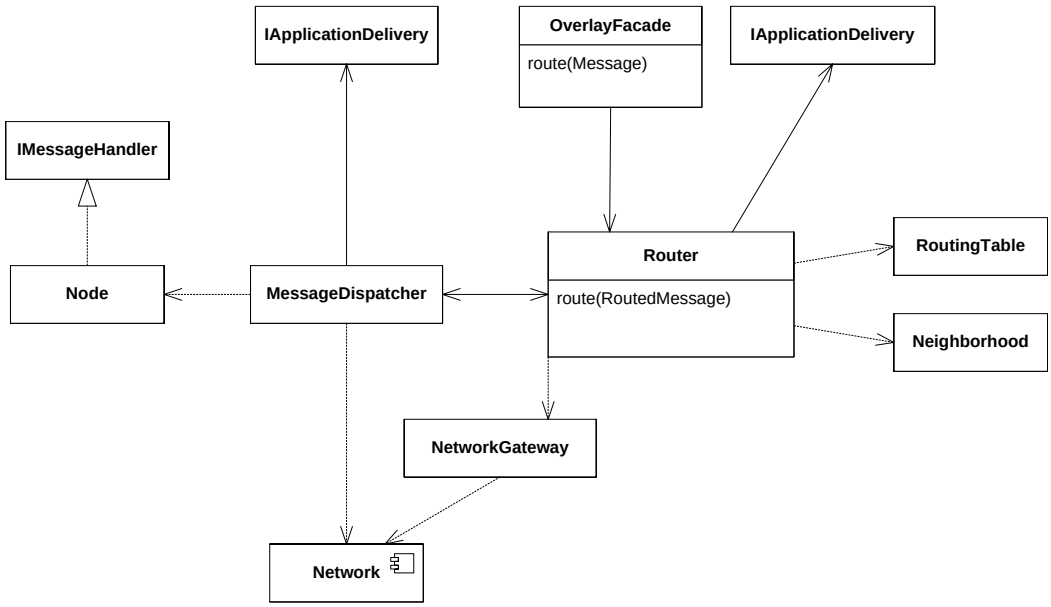
How do you implement the routing algorithm and how does it interact with other objects in the overlay network?

Forces

- All messages, including Routed Messages, enter the overlay network through the Message Dispatcher. However, it is not the task of the Message Dispatcher to implement the routing algorithm.
- A message that has arrived at its target node needs to be delivered up to the application. Otherwise, the message needs to be sent to the next node.
- However, in some specific overlay networks, the message must not be delivered up to the application, but needs to be sent to connected nodes. This is for instance the case in a network where storage nodes are connected to super nodes, but only super nodes participate in the routing algorithm. A super node might need to unwrap the Routed Message and send its contained message off to a connected storage node.
- In some cases, the application built on top needs to be informed before a message is sent to the next node (using Application Notification).
- The specific routing algorithm, as well as the metric used, should be encapsulated from the rest of the overlay network, so that its implementation can be changed easily.
- Despite the complexity of the algorithm, it should be easy to read and understand where the routing takes place in the code.
- The same code should be used when an application uses the overlay network to route a message, and when a Routed Messages arrives at an intermediate node. This avoids code duplication.
- The Message Dispatcher knows whom to dispatch which message. Therefore, once a Routed Message has arrived at its target, the internal message should be dispatched off using the Message Dispatcher. Otherwise, it would lead to code duplication.
- Information about the network (Node Handles) might not only be stored in a routing table, but in a neighbors and other objects as well. This information must potentially also be taken into account in the routing algorithm.

- Not only Basic Messages might be contained in Routed Messages, but Control Messages (e.g. join messages) as well. Therefore, not all messages that arrive at their target need to be delivered up to the application.

Solution

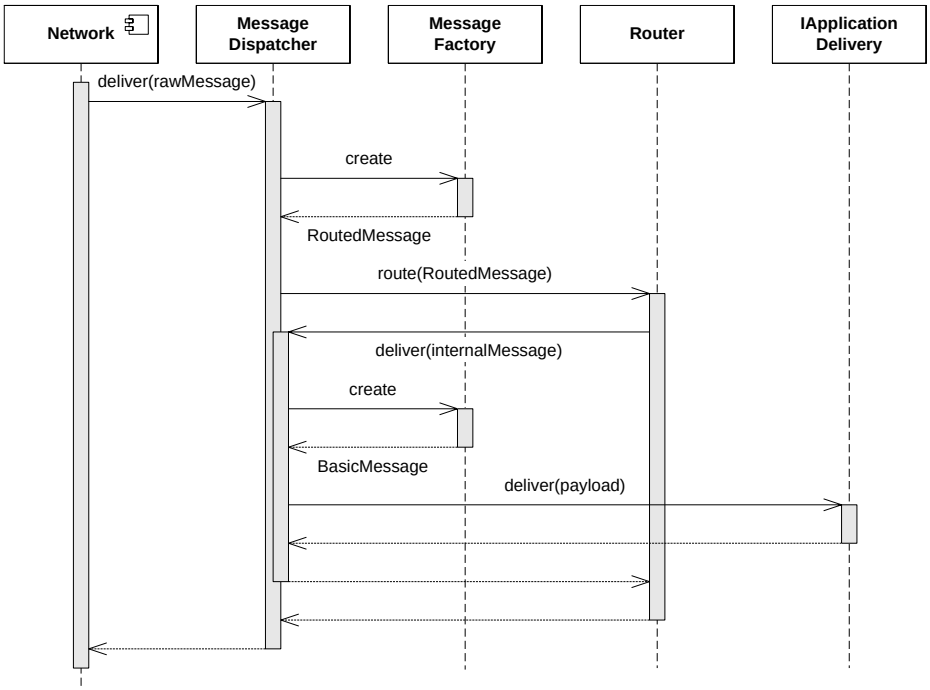


Use a Router, which encapsulates the routing algorithm and provides a method to route Routed Messages. This method checks whether a Routed Message is at its target, or if it needs to be sent away, in which case the router chooses the next node according to its routing algorithm from the routing table and neighbors table. It interacts with the Message Dispatcher to dispatch messages that have arrived at their target to the appropriate object.

The Router is implemented by an object which provides a method *route(RoutedMessage message)*, which either decides that it is at its target, or that it needs to be sent away, in which case it would call the appropriate method on the Network Gateway. Because in the overlay network, not all Basic Messages need to be routed, as well as because not only Basic Messages, but also Control Messages can be contained in Routed Messages, the Message Dispatcher already needs to know whom to dispatch which message. Therefore, the Router should not need to dispatch off messages by itself, because this would result in unnecessary code duplication. Instead, the Router should rely on the Message Dispatcher

to do that job. However, there are some interesting details regarding the actual implementation of the interaction of the Router with the Message Dispatcher.

A simple and elegant solution looks as follows. When a Routed Message arrives at the Message Dispatcher, it dispatches it to the Router (either calling *route* directly in an explicit Message Dispatcher, or let the Router object implement a Message Handler). The Router decides whether the Routed Messages has arrived at its target, or if it needs to be sent away to the next node using the Network Gateway. In case it has arrived at its target, the payload of the Routed Message, thus the actual Basic or Control Message, will be delivered again to the Message Dispatcher. Since the payload is not yet transformed into a message object, delivering it works exactly the same way as when messages are delivered by the network layer. However, it might be necessary for the Message Dispatcher to distinguish these two cases, so that it would need to provide a separate method to the Router, which takes note of it (for instance making a distinction for the Traffic Monitor) and delegates to the usual *deliver()* method. In any case, the Message Dispatcher now lets the message object be created by the Message Factory, and then dispatches it off to the appropriate message as if it was sent directly and arrived through the network layer. The following figures shows a sequence diagram of an example where a Routed Message contains a Basic Message and arrives at its target node.



This is the preferred implementation if this mechanism is sufficient, which is usually true in networks with only one type of node. In networks with several node types, the target node of the routing algorithm might not be the same as the one that should receive the message in the end (e.g. the target of a message is a storage node, but only super nodes take part in the routing process). In this case, the message needs to be sent away to that machine. Even though this could be achieved in the aforementioned implementation, the next approach makes this more explicit and readable for the programmer.

In the second approach, the Message Dispatcher does not immediately dispatch a Routed Message off to the Router, but calls a method *isAtTarget(RoutedMessage routedMessage)* provided by the Router, which returns true, if the Routed Message is at its target, or false otherwise. If so, the Message Dispatcher might either deliver it to the application, or, as motivated, needs to send it off to a connected node (this could be a super node which has connections to several storage nodes). It could call another object which takes care for that. However, this implementation makes the execution flow more explicit. If the message is not at its target, the Message Dispatcher calls the *route()* method from the Router, to actually send it to the next node on the routing path.

Code Samples

```
public class MessageDispatcher {

    public void dispatch(InternetAddress sender, ByteBuffer data) {
        assert sender != null;
        assert data != null;

        Message message = messageFactory.create(data);

        // ...

        if (message instanceof RoutedMessage) {
            router.route((RoutedMessage) message);
        } else if (message instanceof BasicMessage) {
            applicationDelivery.deliver(sender, ((BasicMessage) message)
             .getPayload());
        } else if (message instanceof ControlMessage) {
            // ...
        }
```

```
    }

}

public class Router {

    public void route(RoutedMessage routedMessage) {
        assert routedMessage != null;

        // implement routing algorithm

        // either send to the next node using Network Gateway,
        // or deliver the internal message to the Message Dispatcher
    }

}
```

Resulting Context

The Router is responsible for routing messages to given keys. It therefore expects the messages in a certain format that contains the necessary information. This special message is a Routed Message.

The Router needs to interact with the Message Dispatcher, because it does not know how to dispatch a specific message once it has arrived at its target node. The Message Dispatcher, on the other hand, calls the Router in case a Routed Message arrives. In the case that a message needs to be sent further, the Router makes use of the Network Gateway.

The Overlay Facade uses the Router to route Basic Messages, contained in Routed Messages, to their target. Local Node, Self Maintenance or Separate Protocol might make use of the Router to route Control Messages (e.g. join messages), contained in Routed Messages, to their target.

Some applications might need to be informed whenever the router sends off a message to the next node, which gives the application a possibility to control which messages are sent. This functionality is also defined as the *forward()* method in [27]. For that reason, the Router might need to notify the application using Application Notification.

Rationale

The Router encapsulates the routing algorithm and allows making changes to it easily, without affecting other parts of the code. The Router can be used for incoming Routed Messages, as well as for application messages that need to be routed to their target. Using the Router makes it very easy to understand the design of the overlay network even if the routing algorithm is complex.

The Router resolves most forces stated:

- The routing algorithm is separated from the Message Dispatcher, and completely encapsulated in the Router.
- The collaboration of the Message Dispatcher and the Router makes it possible to include the dispatching logic only once, in the Message Dispatcher.
- Again, the collaboration of the Message Dispatcher and the Router allows to process the message in any way needed.
- The router can inform the application built on top before it sends a message away, using Application Notification.
- The implementation of the routing algorithm can be changed, as long as it remains semantically equivalent and conforms to the interface of the Router.
- Using the Router pattern, it is not necessary to understand the routing algorithm, because it is easy to understand what it does.
- The Router is used for both, arriving external Routed Messages, as well as Routed Messages that need to be sent away for the application built on top.
- Because of the collaboration of the Message Dispatcher and the Router, the Router does not need to know whom to dispatch which message.
- The Router can use all information locally available, such as routing tables and neighborhood structures.
- Again, the Router uses the Message Dispatcher to dispatch off the internal message, once it has arrived at its target node.

The solution has the following favorable qualities:

- Encapsulation: The Router encapsulates all routing logic completely, which makes it easy to change the implementation, as long as it remains semantically equivalent.

- High cohesion: Because the routing logic is encapsulated, the architecture is highly cohesive, putting all routing logic at a single place.
- Abstraction: The Router is a nice abstraction that helps to understand the source code.
- Understandability: It is easy to understand the functionality of the Router even if the specific routing algorithm is not understood in detail.
- Clarity: Using an explicit Router makes the structure of the overlay network very clear.
- Avoids code duplication: Because of the design of the Router, it can be used for incoming messages that need to be routed further, as well as for Basic Messages that the application built on top wants to send.

Known Uses

A Router is used in lots of overlay networks. The actual implementation, however, can look very differently. In FreePastry, for instance, messages arrive at the *PastryNode* (the Message Dispatcher combined with Local Node), which dispatches the Routed Messages off to the Router. This in turn is responsible to set up the next hop using the routing algorithm of Pastry. After that, the message is sent to the *PastryNode* again, which dispatches it off to the Router as before. This time, the next hop has been set, so that the Routed Messages is executed as an Autonomous Message, which lets the Node Handle Proxy representing the next hop receive the message. If the Node Handle Proxy's reference to its *PastryNode* equals the next hop, the Routed Message's internal message is received by the local *PastryNode*. If so, the *PastryNode* this time delivers it up to the application. If not, the Node Handle Proxy is responsible for transmitting the message to the actual computer where the node resides. The complexity of this algorithm in FreePastry partly results from the flexibility provided by the use of Node Handle Proxy in combination with Local Node. Maybe it could be simplified by some minor refactoring.

In HyperCast, the Router is called *Forwarding Engine*, which has its own network adapter to receive messages. In HyperCast, only (and all) application messages (Basic Messages) are routed, so that listening on another port is possible. This makes application messages not arrive at the usual dispatcher, but at the *Forwarding Engine* directly.

Type

Proto-pattern (P).

7.5 Local Node

A computer participating in a peer-to-peer system often corresponds to a node in the network. In some overlay networks, however, a computer can host even more than one node. This is mainly due to load balancing reasons, so that powerful computers can be split into a number of so called *virtual nodes*. I refer to the nodes hosted on a computer as the *local nodes*, as opposed to *remote nodes*, which are hosted on other computers. In real networks, often not all nodes have identical roles, which is because of the heterogeneity of the physical computers that take part in the overlay network. One example is computers behind firewalls or network address translators (NAT), which cannot participate in the routing process. Therefore, different roles such as super, storage, and client nodes are introduced. From a design perspective, this has a number of implications. This section describes patterns dealing about the concept of local nodes and their manifestation in the design of the overlay network.

7.5.1 Local Node

Context

In the overlay network, each physical computer represents a node. However, in some networks, each computer may also host more than one node, but many so called *virtual nodes* to provide a simple mechanism for load balancing. The nodes hosted on one computer are often referred to as *local nodes*, as opposed to *remote nodes* which are hosted on other computers. Even though this concept is very simple, the local nodes can cause some design problems in the object space of the overlay network.

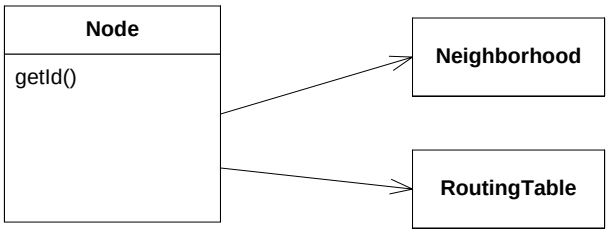
Problem

How do you model the object space of the overlay network around the concept of local nodes?

Forces

- An explicit node object is often not necessary, because it is represented by the physical computer implicitly.
- However, in some cases, several virtual nodes are hosted on the same machine.
- It may be beneficial if all nodes hosted on one machine could profit from shared resources.

Solution



Use a Local Node for each node that is hosted on the computer, resulting in a visual and clear structure in the object space. The Local Node has its own identifier and stores connections to remote nodes (Node Handles), which represent the topology of the overlay network.

Each Local Node is simply represented by an object. This object stores all information about the local node. Connected nodes are stored in the object as references to Node Handles. This makes it easy to see whether a Local Node is disconnected from the network, in which case it can join again.

Resulting Context

Once the Local Node pattern is applied, a lot of design issues are resolved, because it effectively paves the way for future design decisions.

If there are different nodes in your overlay network, you should extend the Local Node by Local Node For Each Type. In case each Local Node can maintain itself in the network by periodically executing a protocol, Self Maintenance can be applied. If the protocols are more complex and need to be executed in different time intervals, a Local Node can be maintained by Separate Protocol instead.

The Local Node should be a Message Handler, so that it processes Control Messages itself.

The connected nodes can be represented by Node Handles.

Rationale

Of course, the solution is very simple and straightforward. However, there are some intricacies which need to be taken into account. The notion of an explicit node is often not necessary, because the physical computer represents a node implicitly. Therefore, objects such as the routing table, for instance, do not need to belong to a specific node, but may reside in a flat object space. When speaking about

the system, messages arrive at nodes. In the software model, however, messages do not arrive at nodes directly, but at the Message Dispatcher. In the case of a Routed Message, the message is processed by the Router and does not even arrive at a Local Node at all. Bringing the system and the software model into close conciliation, however, would not lead to the best solution, because resources could not be shared anymore. The Message Dispatcher and the Router are orthogonal to the Local Node, if several virtual nodes are hosted. Instead of regarding the physical computer as a node in the system, it helps a lot to view the physical computer as simply hosting different nodes.

Note, however, that this discussion is only relevant if all nodes are listening on the same port. If virtual nodes are implemented by nodes listening on different ports, then a more pure, system-oriented design can be applied. However, using different ports is often not the best option for practical peer-to-peer systems.

The Local Node resolves most forces stated:

- Even though an explicit node object is not always necessary, it is much easier to read and understand the code.
- If several virtual nodes are hosted on the same machine, a Local Node object becomes necessary.
- The Local Node objects may not need to be separated strictly. Instead, they can make use of some shared resources, such as the routing table.

The solution has the following favorable qualities:

- Encapsulation: Local Node encapsulates all the logic and data belonging to the local node in a proper entity.
- High cohesion: Because the Local Node builds an encapsulated unit, it leads to high functional cohesion.
- Clarity: Using Local Node results in a very clear and explicit structure of the overlay network, which makes it easy to distinguish local nodes from remote nodes.
- Understandability: It is much simpler to understand the overlay network if the nodes are made explicit.
- Extendibility: Local Node makes it easy to extend the concept of the local node.

Known Uses

FreePastry uses the concept of Local Node, but it extends it heavily. There, the *PastryNode* combines several patterns presented here. The *PastryNode* object implements the Message Dispatcher pattern. Messages directly arrive at the node, instead of a separate Message Dispatcher object. This is feasible because the demultiplexing of which node receives a message is done at a lower level (each node listens on a different network channel). If only one port / network channel should be used, then a separate Message Dispatcher will be needed. But the *PastryNode* object even takes on further responsibilities. In fact, when the application wants to route a message, it is received by the *PastryNode* as if the message came through the network. In FreePastry, each application is built on top of a specific *PastryNode*. When a message arrives for this application, the *PastryNode* uses as variant of ApplicationDelivery to deliver it to the application. It is also possible to register several applications per *PastryNode*, but the reverse is not possible. Therefore, there is only *PastryNode* per application. This is different from other approaches where there is only one application, but several virtual nodes underlying in the overlay network. In fact, this design decisions implies a number of issues. In FreePastry, a message always arrives at a *PastryNode*, which then dispatches it to the appropriate object, such as the Router if it is a Routed Message. The Router sets up the next hope of the message, and now either sends it to the next node using a variant of Node Handle Proxy, or lets the wrapped message be received by the PastryNode (this is combined together by letting each Node Handle Proxy have a reference to a Local Node and vice versa). In this case, the *PastryNode* will deliver the message up to the application.

Type

Standard design solution (S).

7.5.2 Local Node For Each Type

Context

You are using Local Node, but there are different types of nodes in your overlay network, such as super, storage, and client nodes. These different node types also have different properties and behavior, and are connected with different remote nodes in the network. Depending on the resources of the physical computer, different node types need to be hosted.

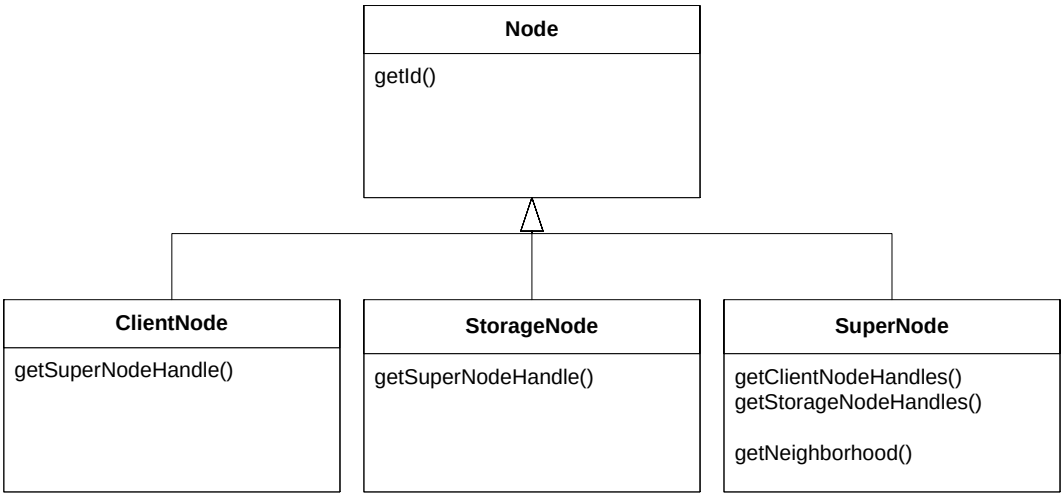
Problem

How do you integrate different node types with different behavior?

Forces

- The different node types might share some properties and behavior.
- Not making the different node types explicit results in code that is hard to maintain because a lot of special cases (e.g. lots of *if* and *switch* statements) are needed.
- Different node types might need to be initialized dynamically on different computers.

Solution



Use Local Node For Each Type, which represents each different node type in the network explicitly. Local Node For Each Type extends the Local Node, which provides the properties and behavior that are shared among all node types.

Client nodes are fundamentally different from super nodes; they have different responsibilities and are connected with different node types in the network. A super node usually fulfills a number of tasks, such as participating in the routing process and maintaining the network topology. Client nodes, on the other hand,

simply rely on a super node and therefore only need to be connected to few other nodes. They also do not participate in the routing process. This different behavior needs to have a manifestation in the code, which is represented by the Local Node For Each Type.

Depending on the physical computer, different Local Nodes can be hosted. A computer behind a firewall or network address translator (NAT) can usually not host a super node, whereas a powerful computer with lots of storage and bandwidth can host several storage nodes (virtual nodes). Using Local Node For Each Type allows makeing this configuration very dynamically, by simply initializing the specific Local Node object.

If you are using Self Maintenance, the new node will join the network automatically and maintains itself up from its creation. The different node types need to perform different maintenance protocols, which are naturally set up by using this pattern.

Resulting Context

Once Local Node For Each Type is applied, the different node types are properly set up. In this case, it can be very useful to use Specialized Message Types and Source Sink Marker for the different messages that are sent among the different node types. This helps to integrate each different node type smoothly with the Message Dispatcher. Therefore, each Local Node should also be a Message Handler. When referring to a specific type of node on another computer, Typed Node Handle should be used. Local Node For Each Type is also the base pattern for Self Maintenance, which implements the maintenance protocol.

Rationale

Using a Local Node For Each Type helps in implementing the different behavior and different properties in a natural way. Because peer-to-peer systems are symmetric, a clear distinction of the different roles is often taken late in the development process, so that it is reflected in different places using *if* and *switch* statements, which do not lead to a nice, object-oriented design. This is the reason to let this simply best practices stand as its own pattern. Of course, using Local Node For Each Type is simply traditional object-oriented modeling, where a Local Node is inherited by special cases.

Local Node For Each Type resolves most forces:

- The different node types can easily share behavior and properties in their common super type.

- Instead of complicated logic in the Local Node object, Local Node For Each Type splits the different node types and allows implementing the special behavior separate.
- The different properties of heterogeneous computers can be reflected by initializing different node types on each computer. Using Local Node For Each Type makes this very easy.

The solution has the following favorable qualities:

- Understandability: Because the different behavior is not implemented in one node object with complicated conditional logic and special cases, the behavior of each node is much simpler to understand.
- Clarity: Local Node For Each Type makes it clear, which nodes are running on each computer.
- Type-safety: Because the Local Node object is split in one for each node type, strong types can be used in each Local Node For Each Type, which leads to stronger type-safety.
- Flexibility: It is easy to configure and run different Local Node For Each Type on each computer.

Known Uses

Local Node For Each Type is simply an example of using inheritance and polymorphism, which is applied in all object-oriented software systems. Unfortunately, in overlay networks, this specific pattern is not often used. Instead, the code is written very 'symmetric', resulting in lots of *if* and *switch* statements along the execution of the code, obfuscating its behavior. Using Local Node For Each Type reflects the different behavior clearly and explicitly.

Type

Standard design solution (S).

7.6 Protocol

In an overlay network, nodes constantly join and leave. Because of these dynamics, the routing topology needs to be maintained in order to guarantee high routing efficiency. These dynamic operations are specified in protocols. In this section, two alternative patterns are described for implementing the protocols.

7.6.1 Self Maintenance

Context

Nodes constantly join and leave the overlay network, which makes it necessary to maintain the routing topology and update it to reflect the changes. Therefore, every node needs to run a maintenance protocol periodically, which sends 'alive' messages to connected nodes, checks if connected nodes are still alive, and much more. If a node realizes that its connected node is not online anymore, it will need to join again.

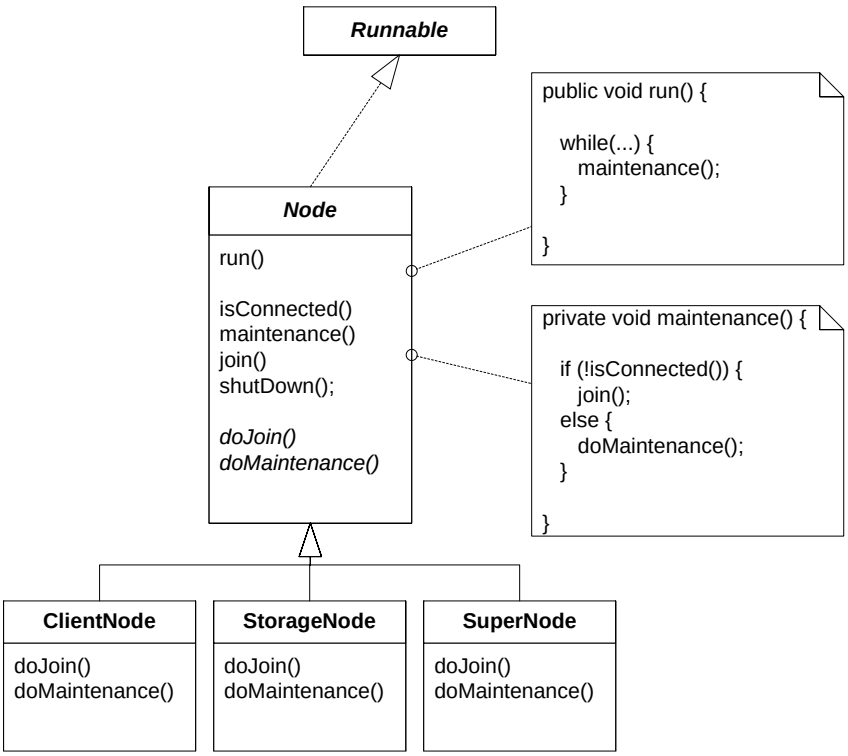
Problem

How do you implement the maintenance protocol?

Forces

- Protocol logic spread around the overlay network is hard to read and maintain.
- The only entities active in the overlay network are the nodes.
- The maintenance protocol needs to be run periodically.

Solution



Use Self Maintenance, which encapsulates the maintenance protocol in the Local Node and runs it periodically in its own thread. This makes the nodes be the only active components in the system, responsible for joining and maintaining themselves the same way as in the system model.

In Java, the Local Node object simply implements the *Runnable* interface. In the *run()* method, the *maintenance()* method of the Local Node object is periodically called in a loop, which waits before looping the next time for a certain amount of time (the maintenance frequency). The *maintenance()* method implements the protocol. It also lets the Local Node join the network if it is not connected. Therefore, it is sufficient to construct the Local Node and start a new thread on it. Instead of implementing the *Runnable* interface, the Local Node could also extend the *Thread* class.

This maintenance protocol represents the periodic action that is performed by each node. Of course, the node also needs to react on Control Messages that arrive. Because Local Node can be combined with Message Handler, also the reactive part of the protocol is encapsulated in the Local Node object. However, the methods

performing the protocol need to be implemented carefully, taking into account that the receiver thread tries to access the same objects at the same time. Therefore, safety and liveness issues must be solved by applying good design principles (see for instance [34]).

The *maintenance()* method does not need to implement the whole logic by itself, but can also rely on helper objects. However, by using Self Maintenance, the protocol is properly encapsulated in the object, and reading the code is very easy.

Code Samples

```
public abstract class Node implements Runnable {

    public void run() {
        while (/* ... */) {
            maintenance();

            try {
                long beforeWait = System.currentTimeMillis();
                long maintenanceFrequency = getMaintenanceFrequency();

                while ((System.currentTimeMillis() - beforeWait)
                  < maintenanceFrequency)) {
                    wait(maintenanceFrequency);
                }
            } catch (InterruptedException e) {
                // ...
            }
        }
    }

    private void join() {
        // ...

        doJoin();
    }

    private void maintenance() {
        if (!isConnected()) {
            join();
```

```
            return;
        } else {
            doMaintenance();
        }
    }

    protected abstract void doJoin();
    protected abstract void doMaintenance();
    protected abstract boolean isConnected();

}

public class ClientNode extends Node {

    private SuperNodeHandle connectedSuperNode; // using Typed Node Handle

    protected void doJoin() {
        // ...
    }

    protected void doMaintenance() {
        // ...
    }

    protected boolean isConnected() {
        return (connectedSuperNode != null);
    }

}

// creating and starting a new node

ClientNode clientNode = new ClientNode();
Thread thread = new Thread(clientNode);
thread.start();
```

Resulting Context

Some events occurring in the maintenance protocol execution might be interesting to the application built on top (e.g. when the Local Node is ready or disconnected

from the network). In this case, Application Notification can be used to inform the application.

Rationale

Using Self Maintenance corresponds to the way overlay networks are being modeled from a system's perspective. It is the nodes that join and leave the network, and it is the nodes, and not protocols, which send 'alive' and other Control Messages. Furthermore, using Local Node as Message Handler plus Self Maintenance encapsulates the Local Node completely.

Local Node resolves most forces stated.

- The protocol logic of each node is encapsulated in the node object itself.
- Using Self Maintenance, only the nodes are active in the system, making it easy to understand the dynamics of the network.
- The Local Node runs its maintenance protocol periodically.

The solution has the following favorable qualities:

- Encapsulation: The dynamics are encapsulated in the Local Node object.
- High cohesion: Because everything is encapsulated in the Local Node object, this leads to high cohesion.
- Low coupling: Because the logic is encapsulated in the Local Node object, there is only low coupling involved when executing the protocol.
- Abstraction: Because the nodes are active in the system, Self Maintenance leads to a nice abstraction.
- Understandability: It is easy to understand where and when each node runs its maintenance protocol.

Unfortunately, the pattern misses the following qualities:

- Flexibility: It is not easy to change the protocol or use different protocols for different tasks. If this is needed, Separate Protocol might be the better choice.
- Extendibility: The protocol cannot be extended that easily, because the whole protocol is regarded as one block, instead of different protocols for different tasks.

Known Uses

Unfortunately, I have not seen this pattern being used in other systems. More often, Separate Protocol was applied. However, in the case that the maintenance algorithm can be triggered at a constant time interval per Local Node, it can be very beneficial.

Alternative Patterns

If different protocols need to be triggered at different time intervals, Separate Protocol can be applied.

Type

Proto-pattern (P).

References

Because of the inherent concurrency, safety and liveness issues need to be solved. Design principles and patterns for concurrent systems are described in [34].

7.6.2 Separate Protocol

Context

You are in a very similar context as in Self Maintenance. However, compared to Self Maintenance, different protocols need to be run at different time intervals.

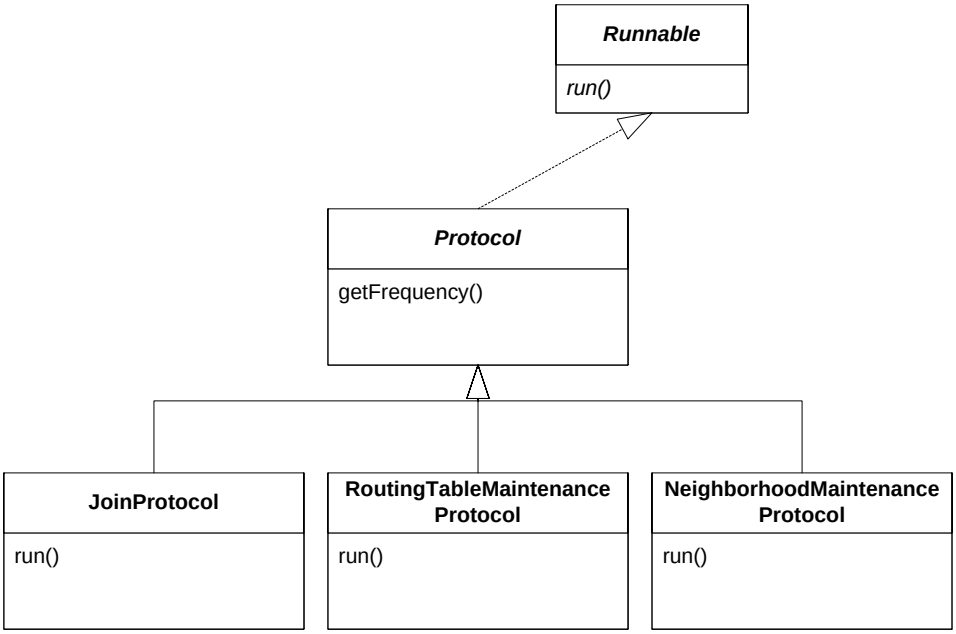
Problem

How do you implement different protocols, needing to run at different time intervals?

Forces

- Nodes are active in the system, not protocols.
- Different protocols need to be executed at different time intervals.
- A single protocol can have quite a complex logic.
- A single protocol might be switched for another one.

Solution



Use a Separate Protocol for each different protocol, encapsulating the respective logic. Each Separate Protocol can be run at different time intervals.

A Separate Protocol corresponds to an object or several objects implementing a specific protocol. For each protocol, a new object is responsible, e.g. a join protocol object, a routing table protocol object, and a neighborhood protocol object. The protocol objects run as their own thread, looping periodically at different time intervals.

Resulting Context

Some events happening in the protocol might be interesting to the application built on top, in which case Application Notification can be used. Such an event could be that the Local Node is disconnected from the network, in which case the application might have more information (e.g. seeds) to bootstrap again.

Rationale

Separate Protocol creates a separate, self-contained entity for each protocol that needs to be run. If the protocols are complex, this pattern nicely separates

the different concerns. Additionally, it allows that each protocol can be run at different time intervals, which is not possible using Self Maintenance. However, from a system's perspective, it is not protocols that are active themselves, but nodes acting according to protocols. At the level of the software system, this notion needs to be traded off for the stated benefits.

Separate Protocol resolves some of the forces stated:

- Compared to Self Maintenance, the node objects are not 'active' any more, but the protocols run separately.
- It is trivial to run the different protocols at different time intervals.
- Each protocol is encapsulated completely and can include a complex logic.
- Each protocol can be switched separately, making it easy to change the configuration and try different protocols individually.

The solution has the following favorable qualities:

- Encapsulation: The Separate Protocol encapsulates the logic of each individual protocol.
- High cohesion: The strong encapsulation makes the design functionally cohesive.
- Flexibility: Each protocol can be replaced individually.
- Extendibility: Each protocol can be extended easily.
- Understandability: Using Separate Protocol, it is easy to understand which entity executes which protocol.
- Abstraction: If an individual protocol is important, it is nice to have a proper abstraction for it.
- Clarity: If each protocol is encapsulated as an entitiy, it results in a clear structure of the protocols.

Known Uses

Most overlay network use this pattern to implement their protocols. FreePastry uses it to implement the *join*, *leafset*, and *routeset* protocol. The latter two are responsible for maintaining the routing topology in the Pastry network.

Alternative Patterns

If there is only one maintenance protocol, Self Maintenance might be a better alternative because it corresponds to the notion of nodes being active in the system, not protocols.

Type

Standard design solution (S).

7.7 Remote Nodes

The local nodes are communicating with remote nodes in the overlay network by exchanging messages. The patterns in this section describe ways to represent remote nodes.

7.7.1 Node Handle

Context

Remote nodes in the network are identified by unique identifiers, but in order to send a message to them, at least also the Internet address (IP address plus port) is needed. Additional information about a node may be necessary at different places in the overlay network, such as for instance the last time an 'alive' message has been received or the proximity of a node in the routing table. Some of these information should be included when node information are transmitted in messages.

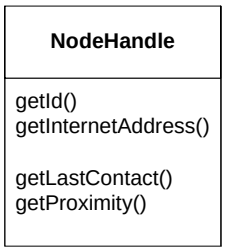
Problem

How do you store all the different information available about a remote node in the overlay network?

Forces

- Not all the information available at one node is needed everywhere.
- Storing the different information about a node at different places makes it hard to maintain and can lead to inconsistencies.
- Storing the different information about a node at different places makes it hard to see which information belongs to a single remote node.
- Some of the information of a node need to be transmitted in messages, but not all of them.

Solution



Use a Node Handle, which provides an abstract handle of a remote node. It stores all available information about a node at a central place. Make the Node Handle serializable or write a proprietary marshalling algorithm, so that the necessary information can be transmitted easily.

The Node Handle is simply represented by an object, with all the information available of a node stored in it. Different parts of the overlay network can then access the needed information through the accessor methods (e.g. *getId()*). Whenever the node information needs to be transmitted, simply use the marshalling method of the object. This can be for instance be by using Java's serialization mechanism (fields that should not be serialized can be made *transient*), or by using a proprietary marshalling method (e.g. *marshall()*) which returns the byte representation of the necessary information of a node.

Resulting Context

The Node Handle can be used wherever a remote node needs to be referenced. That way, each remote node only exists once, so that duplication of information and inconsistencies are avoided by design. If you are having different types of nodes in the network, Node Handle should be extended by Typed Node Handle. Node Handle is also the base pattern for the Node Handle Proxy extension.

Rationale

The solution is very simple and straightforward, and complies with common object-oriented modeling. It avoids duplication of information by encapsulating them in a common object. It also allows extensions of the remote node, which would not have been possible otherwise (e.g. Typed Node Handle, Node Handle Proxy). Because of its simplicity, it is merely worth mentioning. I decided to do so, because it is the base pattern of Typed Node Handle and Node Handle

Proxy, and because some overlay networks do not encapsulate these information properly so that the information is spread around in the overlay network, leading to error-prone and code that is hard to maintain.

Node Handle resolves all forces stated:

- Information about a node can be accessed through the Node Handle object as needed.
- All information about a node is encapsulated in the Node Handle object, leading to consistent data.
- Using Node Handle makes it very clear which remote node is being accessed.
- The Node Handle object can include its own serialization method. This makes it very easy to include it in messages.

The solution has the following favorable qualities:

- Abstraction: The Node Handle is the proper abstraction to refer to remote nodes.
- Encapsulation: All information about remote nodes is encapsulated in the Node Handle.
- Consistency: Because all information about one node only exists at one place when using Node Handle consequently, the data is always consistent.
- High cohesion: All data and methods available for remote nodes are included in the Node Handle object, which leads to high cohesion.
- Extendibility: Using Node Handle, it is easy to extend the concept of remote nodes, for instance by adding a new method or a new data field.
- Understandability: Using Node Handle, it is easy to understand the code, because it is clear what node is refered to at a specific place.
- Clarity: Applying the concept of Node Handle results in a clear structure of the remote nodes.

Known Uses

Node Handle is used in lots of overlay networks, one prominent example being FreePastry, which extends this pattern even further to Node Handle Proxy. Dijjer for example uses a similar pattern to refer to other peers.

Type

Standard design solution (S).

7.7.2 Typed Node Handle

Context

You are using Node Handle to identify remote nodes in the network. However, there are different types of nodes in the network, such as super, storage, and client nodes. Just by a looking at a Node Handle, it is not clear which type of node is stored in it. Apart from that, Node Handles are also not 'type-safe'; it might be possible to add a client node to the routing table, even though they cannot participate in the routing process. Such mistakes are not prohibited by the design, which makes the code error-prone and hard to test. In this case, only super nodes should be possible to add to the routing table. In other situations, however, the exact type of the node does not matter and all Node Handles should be treated alike.

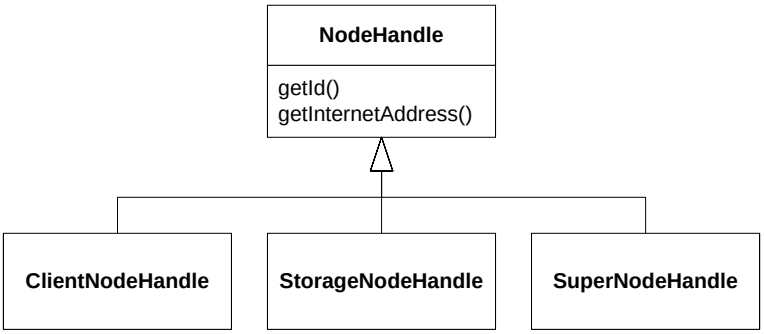
Problem

How can you make the Node Handles 'type-safe', making it easier to detect mistakes and improve readability?

Forces

- Mistakes from using the wrong node type in a Node Handle should already be detected by the compiler (static type-safety).
- It can also be that some node types require more information than others.

Solution



Use a Typed Node Handle, which is simply an extension of a Node Handle for each type of node. Make the base type abstract and consequently use the appropriate Typed Node Handle throughout the overlay network and in message objects.

In detail, the solution looks as follow. Use Node Handle to create a node handle class (*NodeHandle*), which contains common properties such as the identifier and the Internet address of a node, as well as common behavior. Make this node handle class abstract. Then create a class which extends the node handle class for each type (e.g. ClientNodeHandle, StorageNodeHandle, SuperNodeHandle). These classes correspond to Typed Node Handles. Now, consequently use the appropriate Typed Node Handle throughout the overlay network and in message objects, so that always strong types are used (e.g. the routing table only adds SuperNodeHandles, and neighbor messages only take SuperNodeHandles as well).

The specific Typed Node Handle does not need to have properties on its own. It is already beneficial only for the type information. If, however, each node handle has additional properties, or if behavior can be specialized (e.g. at least the *toString()* method can be adapted), then this would be the place to do so. It might be, for instance, that each Typed Node Handle has a special Id object, such as ClientId, StorageId, and SuperId (this is again an example of using strong types).

Whenever the nodes should be treated alike, for instance when iterating through a list of Node Handles, the abstract Node Handle super type can be used.

Resulting Context

Using Typed Node Handle brings a number of benefits. First, type checks can already be done by the compiler. For instance, messages that contain node in-

formation are correctly using a reference to a Typed Node Handle, so that wrong assignments are detected by the compiler. Another example of such static type-safety is the routing table, which does not allow client and storage nodes to be assigned. Altogether, Typed Node Handle renders some unnecessary programming mistakes impossible. Second, reading the source code is getting much easier when using Typed Node Handle. Third, using Typed Node Handle immediately allows extending the remote node handle of each type in different ways. One such extension is for instance Node Handle Proxy.

Rationale

The solution is not really a 'pattern', but simply an example of the use of subtyping (in this case also subclassing) and polymorphism, which are one of the cornerstones of the object-oriented programming paradigm. However, I decided to add it here for the sake of completeness, because using it results in a clearer and less error-prone design. This pattern is a concrete example of the use of subtyping, inheritance and polymorphism.

The Typed Node Handle resolves all forces stated:

- Using Typed Node Handle, refering to remote nodes is type-safe. If the wrong type is used, this can already be detected by the compiler.
- Additional information for special types of nodes can be included in the corresponding Typed Node Handle.

The solution has the following favorable qualities:

- Type-safety: Instead of refering to general Node Types, using Typed Node Handle, the correct node type can always be specified. This makes it type-safe, and wrong assignments can already be detected by the compiler.
- High cohesion: All information about the specific type of a node are grouped together using Typed Node Handle, which leads to high cohesion.
- Encapsulation: The data and behavior of a specific type of node is encapsulated in the Typed Node Handle.
- Abstraction: Typed Node Handle is the right abstraction for different nodes with different roles.
- Clarity: Because of the explicit type of a Node Handle, it is made very clear which remote node type is refered to.

- Understandability: Using Typed Node Handle makes it much easier to understand the source code than if all nodes were refered to in the same way.

Known Uses

Subtyping and polymorphism are employed in all object-oriented software systems. Unfortunately, however, I have not seen this 'pattern' being used in overlay networks for the specific purpose of type-safe Node Handles.

Type

Standard design solution (S).

7.7.3 Node Handle Proxy

Context

You are using Node Handle or Typed Node Handle to refer to a remote node in the overlay network. Whenever a message is sent using the underlying network, it is sent to one of the Node Handles, which means that Extended Overlay Facade is not applied. You want to make it transparent whether the node resides locally or remotely.

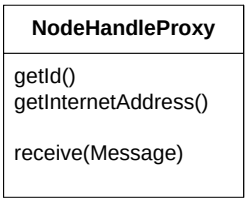
Problem

How can you refer to nodes in the same way, whether they reside locally or remotely, thus making the underlying transmission of a message transparently?

Forces

- Higher abstractions can lead to better designs.
- Referring to local nodes in the same way as remote nodes can be nice.
- Sometimes, however, it can also be good to make the distinction between local and remote nodes explicit.
- Not all aspects can be made transparent, the different latency when contacting local or remote nodes being one example.

Solution



Use a Node Handle Proxy, which can represent both, a remote or a local node. It provides a method *receive(Message message)* which either lets the Local Node process the message or sends the message using the underlying network to the remote node.

If the Node Handle Proxy refers to a Local Node, it stores a reference to that object. If so, *receive(Message message)* will simply call the *handle(Message message)* method of the Local Node object. If not, it will transmit the message to the remote node that it represents (since it stores its identifier as well as its Internet address, it can simply send a message to it).

Using Node Handle Proxy, the Router for example does not need to know whether a node is hosted locally or not.

Resulting Context

The Node Handle Proxy is responsible for transmitting messages to the node it represents. To do so, it will require the service provided by the underlying network layer. In this case, the Node Handle Proxy represents a Network Gateway. Therefore, a Network Stub can be plugged in if needed for testing, and a Traffic Monitor can be informed of each message that is sent using the underlying network.

When using Specialized Message Types and Source Sink Marker, the Node Handle Proxy's *receive()* method could check (e.g. using assertions) whether the sink node type corresponds to the actual Node Handle Proxy type.

Rationale

Node Handle Proxy provides a nice abstraction to communicate with remote nodes, because it hides the underlying transmission of messages. However, compared to remote proxies in other systems, it does not hide the fact that the communication is based on messages, and is therefore a much lower abstraction than other remote proxies. In fact, there are a number of points to consider, which make it arguable whether to use Node Handle Proxy in the first place. First, local

and remote nodes should often not be treated alike. Local nodes, for instance, do not need to be maintained and also do not need to be added to the routing table. Second, it is often necessary to make it explicit, whether a message needs to be transmitted, because it results in message communication, uses bandwidth and has an intrinsic latency. Third, when using Extended Overlay Facade, a Network Gateway is needed anyway, so that using it also to send messages to remote nodes results in a more consistent design.

Node Handle Proxy resolves some of the forces stated:

- Node Handle Proxy builds a higher abstraction about remote nodes. It is not necessary to send messages to remote nodes anymore, but they can actually receive them.
- This allows to treat remote nodes and local nodes alike, making the routing algorithm much more transparent.
- However, this can also have some disadvantages, because accessing local nodes is much cheaper than remote nodes.
- This is very obvious when sending messages to remote nodes, which inherently involves a much longer latency than calling a method on the local node object.

The solution has the following favorable qualities:

- Abstraction: Node Handle Proxy is a convenient abstraction to communicate with remote nodes.
- Encapsulation: Node Handle Proxy encapsulates all logic about the remote nodes properly.

Known Uses

Proxies are used in lots of software systems. Distributed systems often profit from proxy objects because they allow building a higher abstraction and hide the underlying transmission completely (even though not all aspects can be made transparent of course, such as for instance the longer latency when sending messages as opposed to local method calls). However, it is important to note that the abstraction provided by remote proxies is much higher than in Node Handle Proxy, because there methods can be called directly on remote objects, as opposed to sending messages.

Alternative Patterns

If you are using Extended Overlay Facade to also send messages to a known address directly, then a Network Gateway needs to be applied. In this case, it might be more consistent to only use the Network Gateway, instead of Node Handle Proxies.

Type

Adaptation of an existing pattern (A): Proxy.

References

This pattern is very similar to the Proxy pattern described in 'Design Patterns' [19], hence the name. A closely related implementation is used in different distributed systems, such as the built-in remote proxy objects of programming languages such as Java or C#.

7.8 Network Interaction

The overlay network uses the underlying network layer to send and receive messages in a scalable way. This section contains patterns describing the interaction of the overlay network layer with the underlying network layer.

7.8.1 Network Gateway

Context

Incoming messages all enter the overlay network through the single Message Dispatcher. Outgoing messages, on the other hand, leave the overlay network usually from all over the place, simply by calling the *send* operation provided by the underlying network layer. This has a number of disadvantageous. First, it creates a strong dependence on the underlying network layer. High coupling should preferably be avoided, because changes in the network layer potentially affect many places in the code. Swapping the network layer completely with another one will certainly cause a lot of changes in the overlay network. Second, it is hard to put in control mechanisms needing to be executed whenever a message is sent. Such a control mechanism could be a Traffic Monitor that should be informed whenever a messages leaves the overlay network. Doing this at the network layer is not a good idea because first, the network layer does not understand the different message types, and second, it can not be presumed that one can make changes to the network layer (e.g. if using a library).

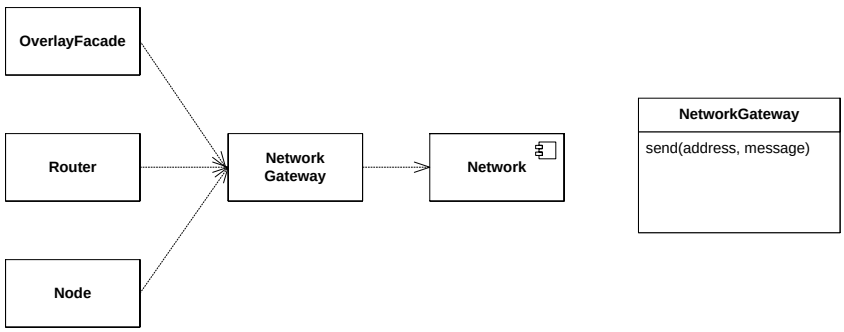
Problem

How do you remove strong dependence on the network layer and allow to put in control mechanisms for outgoing messages?

Forces

- Sending messages using the underlying network layer is very simply, so that it can easily be done wherever a message needs to be sent.
- Different messages need to be sent throughout the overlay network (e.g. router, protocols, and requests by the application).
- Send operations spread throughout the overlay network make it hard to put in control mechanisms that affect all messages that need to be sent, especially if one cannot make changes at the network layer source code.
- The network layer only knows raw bytes and is not the place to reason about message types.
- The network layer may be exchanged by another one, which should not affect a lot of places in the code.
- Testability: Testing the overlay network even without the underlying network layer is not possible of different places in the code rely on a certain *send* operation. Simulating the network is therefore not that easy.

Solution



Use a Network Gateway, which encapsulates access to the underlying network gateway.

The Network Gateway exposes a *send()* method, which simply calls the appropriate method of the underlying network layer. This allows controlling all outgoing messages at a single place, by adding code to this method. It also allows to change the network layer very easily, because then only the Network Gateway is affected by the change, but not all places where messages are actually sent.

Resulting Context

Using the Network Gateway facilitates the use of a Traffic Monitor, because there is a single place which all outgoing messages need to pass. The Network Gateway is also the ideal place to plug in a Network Stub to simulate the network layer.

When using Node Handle Proxy, it can take over the responsibility of a Network Gateway. However, even in this case it might be better to separate the Network Gateway out as an own entity. Even more if Extended Overlay API is applied, in which case not all messages can be sent to known Node Handles, but need to be sent to Abstract Address Handles directly as well.

Rationale

The solution is very simple, but provides a number of benefits. It avoids high coupling between the overlay network and the network layer and provides a single place to extend control mechanism that need to be applied whenever a message is sent through the network (e.g. Traffic Monitor, logging). Additionally, it allows using a Network Stub to plug in a test network layer.

Network Gateway resolves most forces stated:

- Sending messages using the Network Gateway is as simple as calling the underlying network layer directly, once the Network Gateway is properly set up.
- All the different messages can be sent by a simple call to the Network Gateway.
- Using Network Gateway as a single place where messages pass before they are sent off, makes it easy to include control mechanism, such as for instance applying a Traffic Monitor.
- Because the Network Gateway sends message objects rather than raw bytes, it is still possible to deduce message type information.
- Using a Network Gateway makes it trivial to replace the underlying network layer, because only the Network Gateway, but no other place in the code, needs to be adapted.

- The Network Gateway is the ideal place to simulate the sending of messages. This can be done by plugging in a Network Stub at the Network Gateway.

The solution has the following favorable qualities:

- Low coupling: Using a Network Gateway results in minimal coupling between the overlay network layer and the underlying network layer. Only the Network Gateway references the underlying network layer.
- High cohesion: All the logic about accessing the network layer is contained in the Network Gateway, which leads to a highly cohesive architecture.
- Encapsulation: The Network Gateway completely encapsulates the logic about how to access the network layer.
- Extendibility: Besides only sending a message using the underlying network layer, the Network Gateway can also be extended for instance to include a control mechanism before a message is sent (e.g. Traffic Monitor).
- Understandability: It is very easy to understand where messages leave the overlay network.
- Simplicity: Using a Network Gateway, which conforms to the programming style of the whole overlay network, is much simpler than using an external network layer / library directly.

Known Uses

A lot of software systems use gateways when accessing other components, especially if they are provided by third-parties.

In overlay networks, however, Network Gateways are not used that often. Instead, network and the overlay network layer are often melted together. I think that this has a number of drawbacks, because it results in high coupling and a mix of different semantics. Separation of concerns leads to a clearer design which is easier to maintain. The *UdpSocketManager* in Dijjer, however, can be regarded as a variant of a Network Gateway. The Node Handle Proxy approach taken in FreePastry is also a variant of a Network Gateway.

Type

Existing pattern (E): Gateway.

References

Network Gateway is a variant of the Gateway pattern by Martin Fowler, described in 'Patterns Of Enterprise Application Architecture' [37]. Of course, Gateway is similar to other patterns such as Adapter and Facade, described in 'Design Patterns' [19]. However, there are some differences which make Gateway stand as its own pattern; a Facade is usually written by the programmer of the external component to simplify its API for general use, while Adapter is altering an implementation's interface to match another one.

7.8.2 Network Stub

Context

Testing the overlay network is not easy, because it involves a lot of nodes, which may not be available at testing time. Research networks such as PlanetLab exist for exactly this purpose; to provide a test environment for large-scale distributed systems. However, not everyone does have access to such a network, and even if, setting up the environment and deploying the latest source code takes some effort, so that it is not the most practical solution for small changes and frequent tests during (early-stage) development process. Furthermore, the overlay network is dependent on the network layer, so that parallel development of both layers is hindering the testing of the overlay network. Additionally, monitoring the distributed nodes can be very hard if they reside on different computers. In all these cases, it would be beneficial to have a simulated network layer, which can be used whenever one wants to run a test on a small test environment.

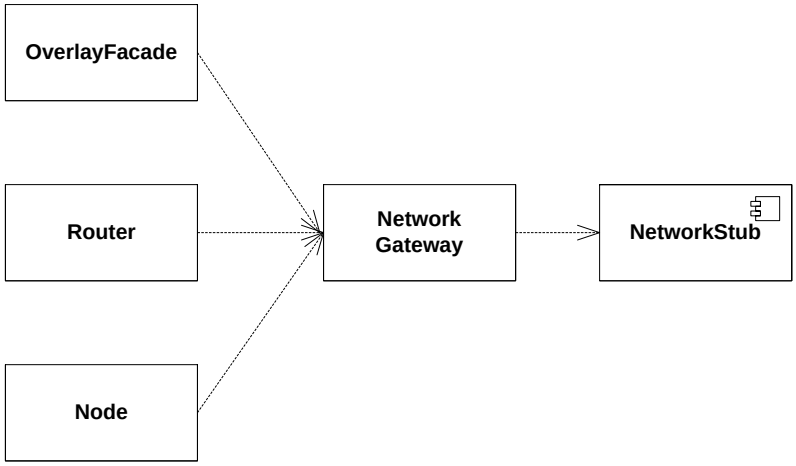
Problem

How can you include a simulation environment for your overlay network?

Forces

- Switching between simulation and real mode should be easy.
- Dependence on the network layer hinders the development process if it is not available.

Solution



Use a Network Stub, which uses the same interface as the network layer, but behaves differently, for instance simulating the sending and receiving of messages.

If you are using Network Gateway, this is the ideal setup to plug a simulation environment into the overlay network. If the Network Stub has the same interface as the network layer, switching between the two modes is easy and possible without affecting the overlay network or the network layer.

Resulting Context

In order to use Network Stub efficiently, the Network Gateway pattern should be applied.

Rationale

Using a Network Stub, it easy to simulate the overlay network with lots of nodes even on a single computer. No changes need to be made to the overlay network, which allows testing the real system as if message were sent using the network layer.

The Network Stub resolves the forces stated:

- Using a Network Stub instead of including the simulation code directly in the Network Gateway, makes it very easy to switch between simulation and real mode.

- The overlay network can be developed even without a network layer in place, because the Network Stub can simulate its functionality.

The solution has the following favorable qualities:

- Encapsulation: The Network Stub encapsulates the simulation logic completely.
- High cohesion: The Network Stub includes all simulation logic, resulting in high cohesion for both, the Network Gateway and the Network Stub.
- Abstraction: The Network Stub is a nice abstraction for a simulated network layer.
- Understandability: It is easy to understand the code and the purpose of the Network Stub.

Known Uses

Stubs are frequently used in software systems which need to access other components. In overlay networks, stubs are for instance used in FreePastry. Although they provide stubs by using special sub types of classes, the basic mechanism remains the same.

Type

Existing pattern (E): Service Stub.

References

Network Stub is simply an example of the Service Stub pattern by David Rice, described in 'Patterns Of Enterprise Application Architecture' [37].

7.8.3 Traffic Monitor

Context

Incoming and outgoing messages often need to be monitored for testing and statistical purposes. Interesting measures include how much bandwidth is used by which message type, how many messages of each type are being sent and received, and many more. This implies that it is often not sufficient to do this monitoring at the network layer, where each message is only considered raw bytes and as such, less semantic information can be extracted.

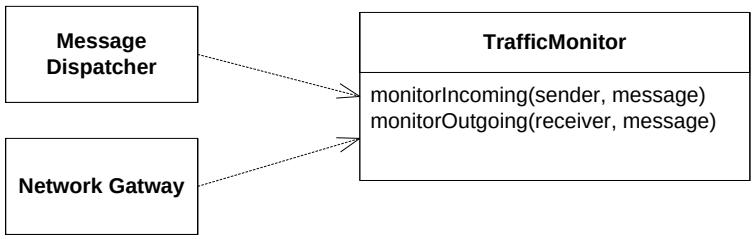
Problem

How can all incoming and outgoing messages easily be monitored?

Forces

- To do interesting measurements, one often needs to have access to the message objects. Therefore, it needs to be done in the overlay network layer.
- The monitoring should be done at a central place, so that changes only affect one place in the source code.

Solution



Use a Traffic Monitor at the overlay network layer, which is informed of all incoming and outgoing messages, interprets them and updates its statistics.

Whenever a message is received or sent, the Traffic Monitor needs to be informed. Receiving messages arrive at the Message Dispatcher. After the Message Factory creates the message object, the Traffic Monitor can be informed of the incoming message (*monitorIncoming()* method). In order to do the same for outgoing messages (*monitorOutgoing()* method), a Network Gateway should best be applied. Another possibility would be to call the appropriate method when a message object is being constructed. This presumes two things. First, there need to be two constructors, one for incoming and one for outgoing messages. Fortunately, this is often given by the fact that one constructor takes parameters while the other one takes raw bytes, creating the object by demarshalling them. Second, all the constructors need to call their super type constructor, so that the call to the Traffic Monitor only needs to be written once. If these requirements are fulfilled, the Traffic Monitor can as well be called by the message constructors. However, it might still be better to do it at the Message Dispatcher and Network Gateway level, because it is the natural place to look and additionally, like that the constructors remain side-effect free.

Resulting Context

The Traffic Monitor does not operate at the network layer and is therefore dependent on other objects to inform it about incoming and outgoing messages. Whenever a message enters the overlay network, the *monitorIncoming()* method needs to be called with the appropriate message object. This can best be done at the Message Dispatcher, after creating the message object using the Message Factory. The same is true for outgoing messages. Whenever a message leaves the overlay network, the *monitorOutgoing()* method needs to be called. If messages can leave at different places, the Traffic Monitor needs to be called everywhere. In order to allow for changes, it is best to apply a Network Gateway and inform the Traffic Monitor from this single place where messages can leave the overlay network.

Rationale

This 'pattern' is straightforward so that it is merely not worth mentioning. However, some overlay networks measure traffic only at the networking layer or do it at different places, so that it is stated here as a 'pattern' simply to document a best practice used in most overlay networks.

The Traffic Monitor resolves all forces stated:

- Because the Traffic Monitor is applied in the overlay network, it is possible to access all message information, such as type, etc.
- The Traffic Monitor centralizes all monitoring activity.

The solution has the following favorable qualities:

- High cohesion: When using a Traffic Monitor, all monitoring logic is put into the corresponding object, instead of having monitoring logic at different places.
- Low coupling: Because the Traffic Monitor takes message objects as arguments and evaluates them autonomously, it results in low coupling between the Traffic Monitor and the rest of the overlay network.
- Avoids code duplication: Because it does the monitoring at a central place, duplicating monitoring logic is not necessary anymore.
- Abstraction: The Traffic Monitor is the right abstraction for all monitoring activity.

- Simplicity: Using the Traffic Monitor is very simple.
- Encapsulation: The Traffic Monitor encapsulates all monitoring logic.
- Understandability: The Traffic Monitor makes it very easy to understand what it does and where the monitoring takes place.
- Clarity: The Traffic Monitor leads to a very clear structure, because it factors the monitoring logic out into a separate entity.
- Reusability: The Traffic Monitor can potentially be reused even for other applications, if it is implemented generically. However, it is arguable whether the needed effort for genericity is justified.
- Extendibility: The Traffic Monitor can be extended easily, if more monitoring capabilities are needed.
- Modularity: The Traffic Monitor leads to a modular structure, which is easy to maintain.

Known Uses

A lot of overlay networks use a solution very similar to Traffic Monitor. P-Grid for instance uses a *Statistics* class for the same purpose.

Type

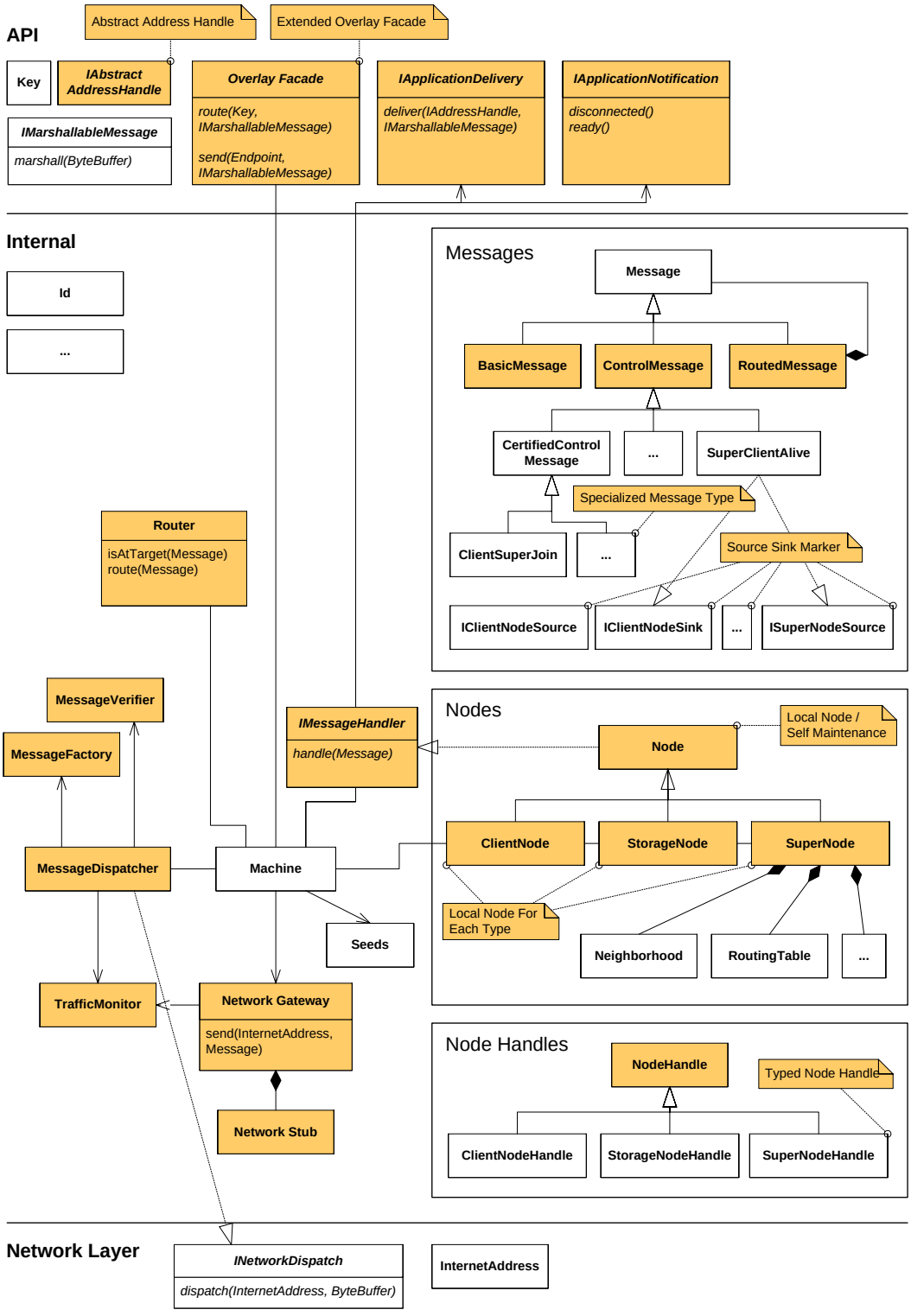
Standard design solution (S).

8 Example

In the last section, each pattern has been described in isolation. In this section, I want to put the patterns together to present a complete example. Where several patterns could be applied to solve a very similar problem, the example has chosen one of the alternatives.

The example is illustrated as a UML class diagram. Together with the explanations in the last section, this should be enough to understand the software architecture in detail. If a class directly corresponds to a pattern, it is highlighted in color. If a pattern is involved which is not described by the class name, a note (also highlighted in color) has been attached to the corresponding class.

Note that the UML class diagram does not completely conform to the standard notation, but is meant as a sketch that illustrates the important information in the simplest form. Of course, the UML class diagram does not show all classes necessary for a running overlay network, but simply represents the classes relating to the patterns, building the skeleton of a software architecture which could be implemented and extended for a real-world project.


API
Abstract Address Handle
Extended Overlay Facade
Key
IAbstract AddressHandle
IMarshallableMessage
marshall(ByteBuffer)
Overlay Facade
route(Key, IMarshallableMessage)
send(Endpoint, IMarshallableMessage)
IApplicationDelivery
deliver(IAddressHandle, IMarshallableMessage)
IApplicationNotification
disconnected()
ready()
Internal
Id
...
Messages
Message
BasicMessage
ControlMessage
RoutedMessage
CertifiedControl Message
...
SuperClientAlive
Specialized Message Type
ClientSuperJoin
...
Source Sink Marker
IClientNodeSource
IClientNodeSink
...
ISuperNodeSource
Router
isAtTarget(Message)
route(Message)
MessageVerifier
MessageFactory
IMessageHandler
handle(Message)
Nodes
Node
Local Node / Self Maintenance
ClientNode
StorageNode
SuperNode
Local Node For Each Type
Neighborhood
RoutingTable
...
MessageDispatcher
Machine
Seeds
TrafficMonitor
Network Gateway
send(InternetAddress, Message)
Network Stub
Node Handles
NodeHandle
Typed Node Handle
ClientNodeHandle
StorageNodeHandle
SuperNodeHandle
Network Layer
INetworkDispatch
dispatch(InternetAddress, ByteBuffer)
InternetAddress

9 Conclusions

In this paper, I have presented a pattern language for overlay networks. A lot of these patterns are known from other software systems, and have simply been adapted to the specific requirements of overlay networks. However, instead of focusing only on new patterns, I have tried to cover the full breadth of design issues in overlay networks. Therefore, I trust that this pattern language can be helpful for both, software designers and programmers new to peer-to-peer systems, as well as for experienced peer-to-peer programmers who will find a collection of familiar ideas generalized as patterns.

The patterns are trying to solve some of the design issues listed in section 5. Therefore, I want to briefly analyze which issues have been solved, and which either still remain open, or have not shown to be actual software design, but rather implementation or system design issues.

- **Interaction with the application:** The application interacts with the overlay network by using an Overlay Facade, or an Extended Overlay Facade. Messages are delivered to the application using Application Delivery, and events are propagated to the application using Application Notification.

- **Routing:** The Router is responsible to encapsulate the routing algorithm. It routes Routed Messages and interacts with the Message Dispatcher. Most of the issues around routing are solved by these patterns.

- **Direct messages:** These problems are addressed by the Extended Overlay Facade, which extends the interface of the overlay network by a *send* operation.

- **Roles:** Roles are properly introduced by Local Node For Each Type. To refer to a specific remote node, Typed Node Handle should be used. Specialized Message Type introduces a new message type between each pair of nodes, which bloats the message hierarchy. For that reason, Source Sink Marker helps to improve readability and lets the Message Dispatcher still remain very simple.

- **Firewall, NAT:** If computers behind firewalls or NATs need to be contacted, different algorithms are implemented. However, Abstract Address Handle can be a good abstraction to refer to remote computers, no matter whether they are behind NATs or not.

- **Message integrity:** Message integrity can be verified by using a Message Verifier. However, other security issues are mostly not a software design, but a system design issue.

- **Message hierarchy:** The complex message hierarchy of overlay networks can be structured into Basic and Control Messages. Routed Message provides a wrapper to route a message in the network. Specialized Message Type increases the number of messages, but allows structuring the message hierarchy more explicit.

- **Message handling:** Several patterns are involved in message handling, such as Message Dispatcher, Message Handler, Autonomous Message, Source Sink Marker, Router and Application Delivery. Most issues regarding message handling are successfully solved by these patterns.

- **Local node:** Local Node and Local Node For Each Type make the notion of local nodes visual and explicit, and allow integrating virtual nodes very easily.

- **Protocol:** Protocols are either implemented by Separate Protocol, or by Local Node together with Self Maintenance.

- **Remote nodes:** Remote nodes in the system are represented by Node Handle, or Typed Node Handle. An even higher abstraction can be applied by using Node Handle Proxy.

- **Execution flow:** The execution flow is made more tractable by different patterns, such as Source Sink Marker, together with an explicit Message Dispatcher, and Self Maintenance. Also Network Gateway helps tracing the execution flow in that it acts as a funnel for all outgoing messages.

- **Testing:** The overlay network can be simulated when using a Network Stub in combination with a Network Gateway. However, testing overlay networks is still complicated. Logging mechanisms and Traffic Monitor can also help for that task.

- **Dynamics:** Using Local Node together with Self Maintenance makes it very clear, where and when the maintenance protocol is executed. Local Node

makes it easy to see the state of the local node and its connectivity. Another way to implement the dynamics is by using Separate Protocol.

According to this list, most of the design issues presented in section 5 are solved by the patterns presented in this paper. However, this list is not comprehensive and other design issues may arise in specific contexts. Furthermore, the patterns only focus on overlay networks, and other patterns may be discovered in a broader scope of peer-to-peer systems.

The contributions of this paper are twofold: First, the pattern language documents all existing and adpated design patterns for most aspects of overlay networks. Second, it presents a number of suggested proto-pattern, which remain to be scrutinized by others.

As future work, more projects need to be investigated to find real evidence for these patterns, and the pattern style needs to be improved. For this reason, it would also be interesting to take part in a workshop for pattern writers and incorporate feedback by others.

Bibliography

[1] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. Technical Report TR-819, MIT, 2001.

[2] P. Druschel and A. Rowstron. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 18 IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[3] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 2003.

[4] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proceedings of IPTPS02*. Cambridge, USA, March 2002.

[5] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. In *Proc. ACM SIGCOMM*, 2001.

[6] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Pattern-Oriented Software Architecture, Vol. 2, New York: John Wiley, 2000.

[7] Project JXTA. *http://www.jxta.org/*

[8] A. Langley. The Trouble with JXTA, *O'Reilly OpenP2P.com: http://www.openp2p.com/pub/a/p2p/2001/05/02/jxta_trouble.html*, May 2001.

[9] D. Lea. Patterns-Discussion FAQ. *http://gee.cs.oswego.edu/dl/pd-FAQ/pd-FAQ.html*, November 2000.

[10] C. Shirky. What Is P2P...And What Isn't? *O'Reilly OpenP2P.com: http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html*, November 2000.

[11] Skype. *http://www.skype.com/.*

[12] Jabber. *http://www.jabber.org/*

[13] Gnutella. *http://www.gnutella.org/*

[14] J. Ritter. Why Gnutella Can't Scale. No, Really. *http://www.darkridge.com/ jpr5/doc/gnutella.html*, February 2001.

[15] K. Aberer, and M. Hauswirth. An Overview on Peer-to-Peer Information Systems. *Workshop on Distributed Data and Structures*, Paris, France, 2002.

[16] D. S. Milojicic, V. Kalogeraki, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu: Peer-to-Peer Computing. *HP White paper*, March 2002.

[17] A. Oram: Peer-to-Peer: Harnessing the Power of Disruptive Technololgies. *O'Reilly*, February 2001.

[18] L. Rising. Pattern Mining. *CRC Handbook of Object Technology, found at http://members.cox.net/risingl1/articles/mining.htm*, 1997.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley*, 1994.

[20] Hillside.net. *http://www.hillside.net/.*

[21] C. Alexander, S. Ishikawa, M. Silverstein with M. Jacobson, I. Fiksdahl-King, S. Angel: A Pattern Language - Towns-Buildings-Constructions. *Oxford University Press*, 1977.

[22] C. Alexander. The Timeless Way of Building. *Oxford University Press*. New York, 1979.

[23] B. Appleton. Patterns and Software: Essential Concepts and Terminology. *http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html*, 1998.

[24] EuroPlop. *http://www.hillside.net/europlop/.*

[25] E. Chtcherbina, and M. Völter: P2P Patterns - Results from the EuroPlop 2002 Focus Group. *http://www.voelter.de/data/pub/P2PSystems.pdf*, December 2002.

[26] S. Herden, and A. Zwanziger. A Pattern-Language for Peer-to-Peer Networks. *Focus Group at EuroPlop 2005, found also directly at http://finglas.cs.uni-magdeburg.de/europlop/*, May 2005.

[27] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured P2P Overlays. *IPTPS'03*, Berkeley, CA, February 2003.

[28] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles*, 2001.

[29] P. Druschel, and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. *HotOS VIII*. Schoss Elmau, Germany, May 2001.

[30] A. Rowstron, A-M. Kermarrec, M. Castro, and P. Druschel. Scribe: The design of a large-scale event notification infrastructure. *NGC2001*, UCL, London, November 2001.

[31] J. Kubiatowicz, et al. OceanStore: An Architecture for Global-Scale Persistent Storage. *ASPLOS*, 2000.

[32] J. Liebeherr, J. Wang, and G. Zhang. Programming Overlay Networks with Overlay Sockets. *NGC 2003 proceeding*, Munich, Germany, September 2003.

[33] HyperCast. *http://www.cs.virginia.edu/ mngroup/hypercast/*.

[34] D. Lea, Concurrent Programming in Java: Design Principles and Patterns, *Second Edition, Addison-Wesley*, 2000.

[35] G. Hohpe, and Bobby Woolf. Enterprise Integration Patterns. *Addison-Wesley*, October 2004.

[36] W. Cunningham. Portland Pattern Repository. *http://www.c2.com/cgi/wiki*

[37] M. Fowler. Patterns of Enterprise Application Architecture. *Addison-Wesley*, 2003.

[38] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. Pattern-Oriented Software Architecture: A Pattern System. *Addison-Wesley*, Boston, 1996.

[39] D. Grolimund, and L. Meisser. Kangoo - A Simple, Robust, and Efficient DHT for Large Amounts of Data. *Semester Project, Distributed Systems, ETH Zurich*, February 2005.

[40] FreePastry. *http://freepastry.rice.edu/FreePastry/*

[41] Tapestry. *http://www.cs.ucsb.edu/ ravenben/tapestry/*

[42] Bamboo. *http://bamboo-dht.org/.*

[43] P-Grid. *http://www.p-grid.org/.*

[44] A Viceroy implementation. *http://www.ece.cmu.edu/ atalmy/viceroy/.*

[45] LimeWire. *http://www.limewire.org/.*

[46] jMule. *http://jmule.sourceforge.net/.*

[47] Dijjer. *http://dijjer.org/.*

[48] OogP2P. *http://www.duke.edu/ cmz/p2p/.*

[49] GISP. *http://gisp.jxta.org/.*

[50] Azureus. *http://azureus.sourceforge.net/*

[51] JTorrent. *http://sourceforge.net/projects/jtorrent/.*

[52] Meteor. *http://meteor.jxta.org/.*

[53] Chord. *http://pdos.csail.mit.edu/chord/.*

[54] Pastry web-site. *http://research.microsoft.com/ antr/Pastry/.*

[55] eMule. *http://www.emule.org/.*

[56] Bunshin. *http://ants.etse.urv.es/bunshin/index.html.*

[57] OpenDHT. *http://www.opendht.org/.*

[58] Pattern Forms. *http://c2.com/cgi-bin/wiki?PatternForms*, April 2005.

[59] Hillside Group. Writing Patterns. *http://www.hillside.net/patterns/writing/writingpatterns.htm*, 2005.

[60] G. Meszaros, and J. Doble. A Pattern Language for Pattern Writing. *http://hillside.net/patterns/writing/patternwritingpaper.htm*,

[61] P2P Sockets for JXTA. *http://p2psockets.jxta.org/.*

[62] B. King. Simplify Distributed System Design Using the Command Pattern, MSMQ, and .NET. *MSDN Mag, found at http://msdn.microsoft.com/msdnmag/issues/04/09/CommandPattern/*, September 2004.

[63] M. Fowler. Layering Principles. *http://www.martinfowler.com/bliki/ LayeringPrinciples.html*.

[64] Marker Interface Pattern. *Wikipedia: http://en.wikipedia.org/wiki/ Marker_interface_pattern*.

[65] Anti-Pattern. *http://en.wikipedia.org/wiki/Antipattern*.

[66] PlanetLab. *http://www.planet-lab.org/*.