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

Dominik Grolimund
dominik.grolimund@inf.ethz.ch

Peter Müller
peter.mueller@inf.ethz.ch

Department of Computer Science
ETH Zurich

Abstract

Peer-to-peer systems typically operate in large-scale, highly unreliable and insecure environments. Tackling this complexity requires good software design. Yet, many peer-to-peer systems are developed in an ad-hoc manner, and little has been published about their software architecture. We studied various academic and open source peer-to-peer systems and identified design patterns for the overlay network, the key architectural component of a peer-to-peer system. In this paper, we present a pattern language for overlay networks, consisting of new patterns as well as adaptations of existing patterns. This language proved highly useful for the development of our own peer-to-peer system.

Note: This paper covers a whole pattern language rather than an individual pattern. For a writers' workshop session, we suggest the patterns *Message Verifier* and *Router*.

Contents

1 Introduction **3**

2 Related Work **5**

3 Overview **7**
3.1 Peer-to-Peer Systems 7
3.2 Overlay Network 7
3.3 Software Architecture 8
3.4 Overlay Network Layer 8
3.5 Design Issues 10
3.5.1 Application Interaction 10
3.5.2 Messages 10
3.5.3 Message Handling 10
3.5.4 Routing 11
3.5.5 Local Nodes 11
3.5.6 Protocol 11
3.5.7 Remote Nodes 11
3.5.8 Network Interaction 11

4 Pattern Language **13**
4.1 Application Interaction 13
4.2 Messages 14
4.3 Message Handling 15
4.4 Routing 16
4.5 Local Nodes 16
4.6 Protocol 17
4.7 Remote Nodes 18
4.8 Network Interaction 18
4.9 Overview 19

5 Proto-Patterns **20**
5.1 Application Interaction: Abstract Address Handle 20
5.2 Messages 22
5.2.1 Message Hierarchy 22
5.2.2 Routed Message 25
5.2.3 Specialized Message Type 27

5.2.4 Source Sink Marker 30
5.3 Message Handling: Message Verifier 33
5.4 Routing: Router 35
5.5 Protocol: Self Maintenance 40

6 Conclusions **44**

Bibliography **44**

1 Introduction

Peer-to-peer systems have gained a lot of popularity due to file sharing applications (e.g., Napster, Gnutella, eDonkey, Kazaa), communication applications (e.g, Skype, Jabber, ICQ), collaboration applications (e.g., Groove Networks), content distribution systems (e.g., BitTorrent), and grid computing (e.g., SETI@Home). The term *peer-to-peer* refers to a decentralized architecture in which all nodes (computers) have identical capabilities and responsibilities and all communication is potentially symmetric [1, 2, 3]. 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 a peer-to-peer system is challenging. While early peer-to-peer applications were developed rather experimentally, they have attracted a great deal of attention from research ever since, resulting in a number of ongoing projects at leading universities around the world. Most efforts have been put into 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 [4], Pastry [5], Tapestry [6], Kademlia [7], CAN [8]), which solve this task efficiently.

So far, research on peer-to-peer systems has mainly focused on system design. However, the complexity of a peer-to-peer system also requires a sophisticated software design. Yet, many existing peer-to-peer applications are developed in an ad-hoc manner, and little has been published about their software architecture. Some design solutions for traditional distributed systems [9] can be adapted to peer-to-peer systems. However, peer-to-peer systems exhibit a number of characteristics that are very different from centralized, asymmetric distributed systems and that must be reflected in the software design. This observation is also supported by peer-to-peer framework initiatives (such as Sun's JXTA [10]), which are building higher abstractions around the core problems of peer-to-peer systems.

Design patterns capture successful solutions to recurring problems and are used both to document and to improve the design of software systems [11, 12, 13, 14]. In this paper, we describe patterns for overlay networks, the key architectural component of a peer-to-peer system, which is responsible for implementing the routing algorithm. To identify the patterns, we investigated different academic and open-source projects such as FreePastry [15], Tapestry [16], Bamboo [17], P-Grid [18], a Viceroy implementation [19], JXTA [10], LimeWire [20], jMule [21], Dijjer [22], OogP2P [23], GISP [24], Azureus [25], JTorrent [26], and Meteor [27]. We also implemented and improved found design patterns in our own peer-to-peer system.

The contributions of this paper are twofold. First, it presents a complete pattern language for overlay networks, consisting of simple new patterns as well as adaptations of existing patterns. Second, it suggests several proto-patterns, which have not yet been documented, but belong to the fundamental building blocks of overlay networks.

This paper is structured as follows. In Sec. 2, we discuss related work. Sec. 3 provides an overview over peer-to-peer systems and the overlay network layer as its core building block, and outlines the basic design issues. The next two sections present our pattern language. We summarize our pattern language in Sec. 4, and provide a detailed description of the proto-patterns we suggest in Sec. 5. Finally, we offer some conclusions in Sec. 6.

2 Related Work

In this section, we discuss related work on the software design of peer-to-peer systems.

To the best of our knowledge, no patterns for peer-to-peer systems have been documented yet. EuroPLoP 2002 [40] hosted a focus group on patterns in peer-to-peer systems, which outlined the characteristics of peer-to-peer systems and concluded that new patterns might be discovered, but that most issues could be solved with well-known patterns for distributed systems. This workshop did not result in the documentation of new patterns. EuroPLoP 2005 organized a follow-up focus group on peer-to-peer systems, but the results have not been published yet.

Dabek *et al.* [41] recognize overlay networks as the key component of most peer-to-peer systems. They show how different higher-level abstractions such as distributed hash tables, decentralized object location, and group multicast can be built on top of overlay networks. These abstractions are in turn the basis for applications such as CFS [34], PAST [35], Scribe [42], and OceanStore [36].

Lieberherr *et al.* [43] present a socket-based approach for the implementation of overlay networks. The overlay socket API describes how applications can interact with different overlay network protocols, and how different transport protocols can be used. Lieberherr *et al.* focus on this interaction rather than on the design of the overlay network itself.

JXTA [10] is a major effort in building a higher abstraction around the core problems of peer-to-peer systems. It consists of a set of protocols that allow any device to communicate 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, etc.

Besides the overlay network, a fully functional peer-to-peer system contains several other important components. However, good design solutions for problems at the application level are well-known and typically not specific to peer-to-peer applications. On the other hand, a rich set of patterns from distributed systems in general is available for problems arising at the network layer. We conclude this section by giving references to these design solutions.

- **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. Lea [44] describes a number of best practices and design patterns for concurrent systems. Additionally, Schmidt *et al.* [9] provide solutions for designing scalable concurrent distributed systems.

- **Messaging:** Nodes in peer-to-peer systems communicate by exchanging messages. Messaging has been investigated extensively in the light of enterprise application integration. The book by Hohpe and Woolf [45] focuses on design issues and patterns.
- **Asynchronous operations:** Sending messages is inherently asynchronous, which results in a number of design problems when offering these services to an application. These problems occur at the application level or the thin layer of distributed hash tables because these layers build a higher abstraction further away from the notion of messages. Design patterns for asynchronous systems are, for instance, presented by Schmidt *et al.* [9].

3 Overview

3.1 Peer-to-Peer Systems

Shirky [28] defines peer-to-peer as: *"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 regards peer-to-peer systems as an application-level Internet on top of the Internet. Peer-to-peer systems consist of a large number of nodes that communicate by requesting and sending data. Of course, this concept is not new, and large parts of the Internet infrastructure itself communicate in a peer-to-peer fashion. What is new is the fact that the nodes in the system are at the edge of the Internet, that is, unreliable personal computers, which can be turned on and off at any time. Peer-to-peer systems therefore need to cope with this inherent dynamics and the high exposure to attacks and failures. Moreover, the scale of peer-to-peer systems can be extremely large, incorporating millions of nodes.

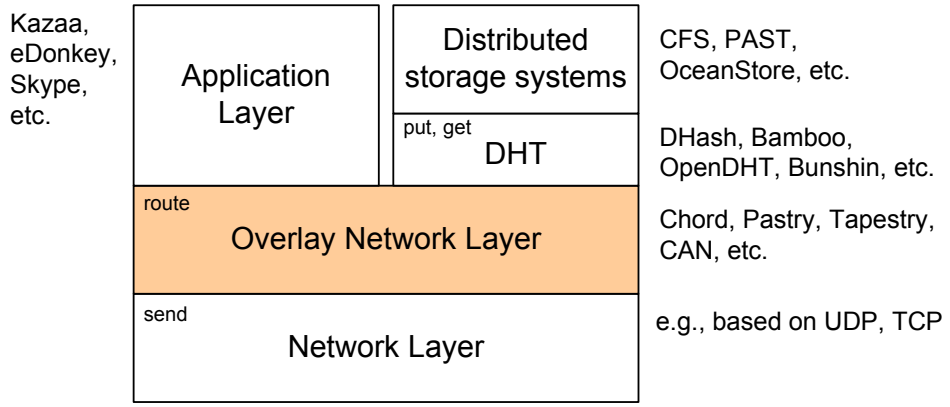
3.2 Overlay Network

Peer-to-peer applications have been built for different purposes such as file sharing (e.g., Napster, Gnutella, eDonkey, Kazaa), communication (e.g, Skype, Jabber, ICQ), collaboration (e.g., Groove Networks), content distribution (e.g., BitTorrent), and grid computing (e.g., SETI@Home). Although their purpose is different, they all share the need to localize items such as fragments, files, and users in the network. Napster, one of the first file sharing applications, used a central server for the localization, which does not scale well. Early systems that followed Napster, such as Gnutella [31], 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 a large number of messages, which is not efficient [32].

Routing efficiency is improved significantly by structured overlay networks such as Chord [4], Pastry [5], Tapestry [6], Kademlia [7], and CAN [8]. Overlay networks 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 node on its local routing table that is closest (using an appropriate metric) to the specified target identifier.

3.3 Software Architecture

Applications are built on top of the overlay network layer, which in turn uses the network layer to transmit messages over the network (e.g., using UDP or TCP). Therefore, the software architecture of a peer-to-peer application typically looks as follows.



Different kinds of applications can be built on top of the overlay network layer, such as file sharing applications [33], collaboration applications [29, 30], or distributed file systems [34, 35, 36]. Distributed file systems, however, are usually not directly built on top of the overlay network layer, but on top of a distributed hash table (DHT) [17, 37, 38, 39], which provides a higher abstraction by offering *put* and *get* operations to store and retrieve data fragments in the network. In the literature, distributed hash tables and overlay networks are sometimes used interchangeably, because distributed hash tables rely on overlay networks to localize the fragments in the network. However, in this paper, we 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 figure above.

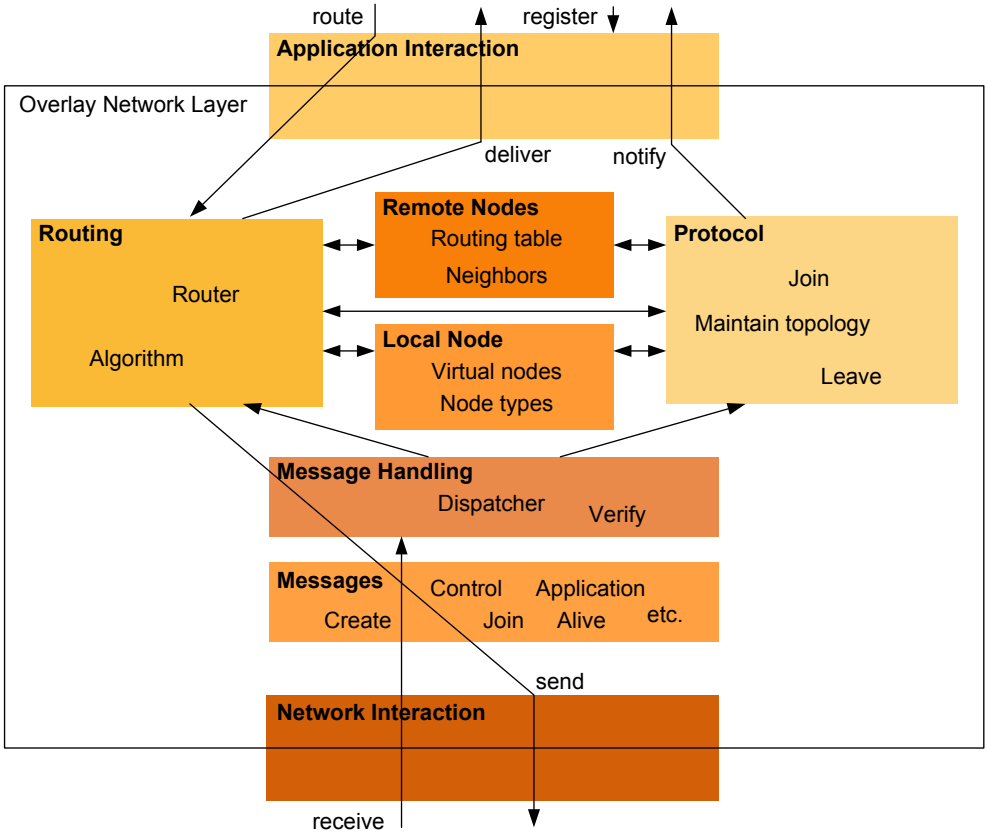
3.4 Overlay Network Layer

The overlay network layer exposes an operation *route*, which is used by applications to route messages to a specified target key in the network. The routing algorithm determines the node with the closest identifier known locally and sends the message to it using the underlying network layer. When the overlay network layer receives a message from the network layer, it checks whether the message has arrived at its target, or whether it needs to be forwarded to the next node on the routing path. Finally, when the message arrives at its target, the overlay network layer delivers it to the corresponding application.

The routing algorithm presumes that a topology is maintained among nodes in the network, which is also the responsibility of the overlay network layer. For that reason, it needs to execute a protocol which maintains neighbors by periodically sending messages. It needs to replace nodes that have left and integrate new nodes that have joined the network. In a specific network, different kinds of node types may exist. Therefore, different kinds of messages may need to be exchanged among these different nodes.

The overlay network layer actually represents a node in the network. However, in some networks, a computer may host more than one node concurrently, so that the overlay network can represent several virtual local nodes.

The different tasks and functions of the overlay network layer can be grouped into different categories as follows. We will use these categories throughout the rest of this paper.



- **Application Interaction** represents the interface of the overlay network layer to applications. It needs to provide a *route* operation and means to register different kinds of applications.
- **Messages** groups all different kinds of messages that are sent in the overlay network.
- **Message Handling** regards the question of how the overlay network processes the different kinds of messages.
- **Routing** combines all parts related to the routing algorithm.
- **Local Nodes** represents the nodes hosted on the computer.
- **Protocol** groups all parts responsible for maintaining the topology.
- **Remote Nodes** organizes how remote nodes are represented in the overlay network, and how interaction takes place.

- **Network Interaction** defines how the overlay network layer interacts with the network layer.

3.5 Design Issues

In this section, we identify different design issues that arise in the design of the overlay network. We group the different issues into the categories introduced in the last section. These design issues build the basis for the patterns of the pattern language presented in this paper, which have proven very helpful for the design of our own overlay network.

3.5.1 Application Interaction

The application built on top of the overlay network layer 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?

3.5.2 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. Furthermore, lots of different kinds of messages exist in the overlay network. There are application 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?

3.5.3 Message Handling

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. Therefore, the following design issues have to be addressed: How are such attacks detected and damage prevented? How can the message integrity be verified?

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.

3.5.4 Routing

Query messages need to be routed to their target. On their way, they pass many 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?

3.5.5 Local Nodes

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*. We refer to the nodes hosted on a computer as the *local nodes*, as opposed to *remote nodes*, which are hosted on other computers. Research networks often only have one type of node, which is responsible for various tasks. In real systems, however, this is rarely feasible. Different roles for the nodes need to be introduced because 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 is 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?

3.5.6 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. If computers crash, they cannot properly leave the system. Still, the system needs to cope with these failures. These dynamic operations are specified in protocols.

3.5.7 Remote Nodes

In the overlay network, the local nodes communicate with remote 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.

3.5.8 Network Interaction

The overlay network uses the underlying network layer to send and receive messages in a scalable way. How does the overlay network interact with the network layer? The overlay

network layer should not depend on the actual implementation of the network layer, so that it can be exchanged at any time. Furthermore, the actual transport protocol such as UDP or TCP should not need to be exposed to the overlay network layer.

4 Pattern Language

This section presents our 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, and nodes, trying to solve most of the design issues presented in section Sec. 3.5. The language does not cover the underlying network layer, which transmits data over the network.

The pattern language consists of adaptations of well-known patterns as well as new proto-patterns that represent best-practices that have led to a favorable design in different projects. Combining the patterns appropriately will result in a functional skeleton of an overlay network. An example can be found in [46, Sec. 8].

For each pattern, we summarize the problem it addresses and sketch the solution. The suggested proto-patterns will be presented in detail in Sec. 5. We have grouped the patterns into the eight categories introduced in Sec. 3.2. The patterns in each category are separated into 'known and adapted patterns' and 'proto-patterns'. In case the pattern already exists or is an adaptation from a well-known pattern, the original pattern is stated in parenthesis.

4.1 Application Interaction

Known or Adapted Patterns

Pattern	Problem	Solution
Overlay Facade (Facade)	How do you encapsulate access to the overlay network from the application? How are direct messages sent? Are they sent using the overlay network, or by accessing the underlying network layer directly?	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. Extend the Overlay Facade with a *send* method, and use the overlay network layer also to send direct messages to known addresses.
Application Delivery (Observer)	How does the overlay network deliver messages up 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)	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.

Proto-Patterns

Pattern	Problem	Solution
Abstract Address Handle	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.

4.2 Messages

Known or Adapted Patterns

Pattern	Problem	Solution
Message Factory (Factory)	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)	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.

Proto-Patterns

Pattern	Problem	Solution
Message Hierarchy	How do you separate messages sent by the application from messages sent by the overlay network?	Use Message Hierarchy to structure the messages clearly into application and control messages. Application messages wrap messages sent by the application in a simple Envelope Wrapper. An application message can be as simple as only containing the payload. The pattern allows the Message Dispatcher to distinguish application from control messages sent by the overlay network.
Routed Message (Envelope Wrapper)	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	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 Interface)	How can you make the source and sink node type of a message explicit, so that it can be checked at runtime 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 by the program.

4.3 Message Handling

Known or Adapted Patterns

Pattern	Problem	Solution
Message Dispatcher (Observer)	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)	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)	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.

Proto-Patterns

Pattern	Problem	Solution
Message Verifier	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.

4.4 Routing

Proto-Patterns

Pattern	Problem	Solution
Router	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.

4.5 Local Nodes

Simple Patterns

Pattern	Problem	Solution
Local Node	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	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.

4.6 Protocol

Known or Adapted Patterns

Pattern	Problem	Solution
Separate Protocol (Strategy)	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.

Proto-Patterns

Pattern	Problem	Solution
Self Maintenance	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.

4.7 Remote Nodes

Simple or Known Patterns

Pattern	Problem	Solution
Node Handle	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 marshaling algorithm, so that the necessary information can be transmitted easily.
Typed Node Handle	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)	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.

4.8 Network Interaction

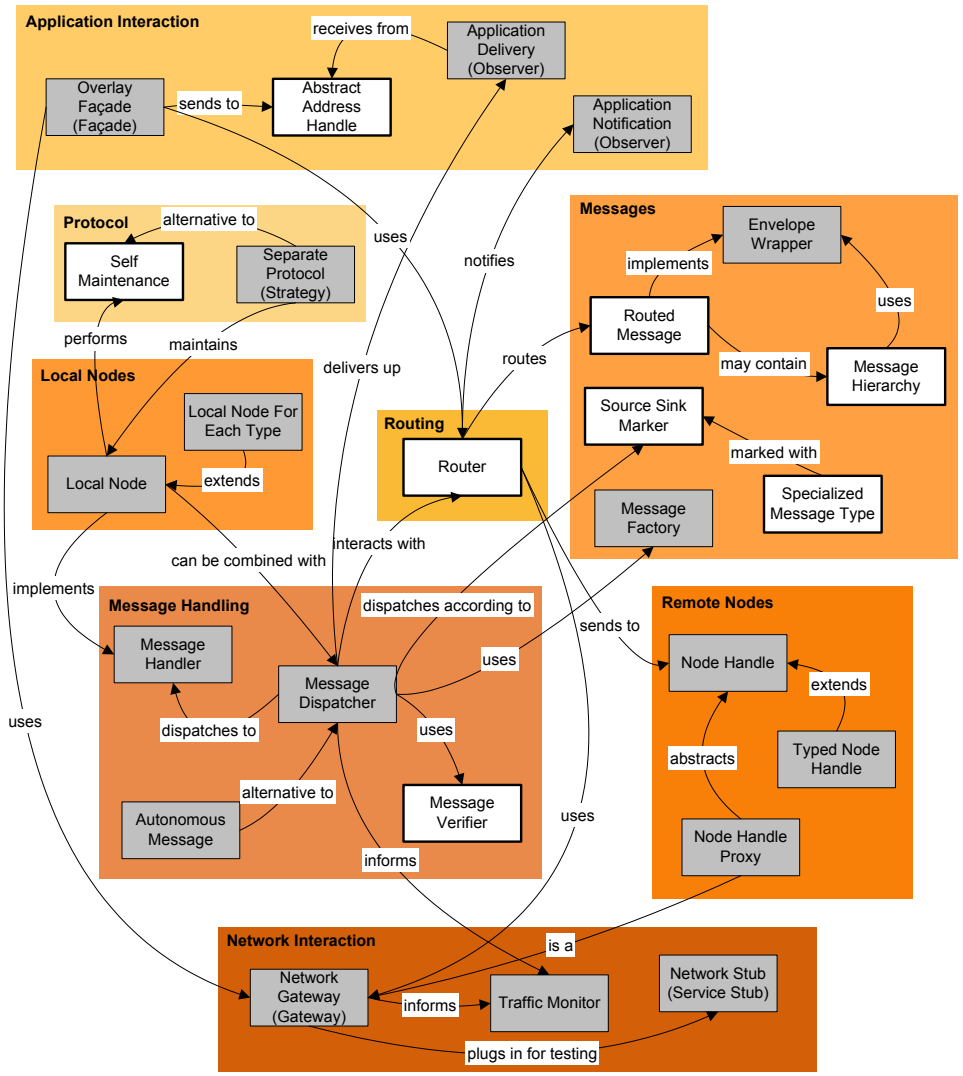
Simple or Known Patterns

Pattern	Problem	Solution
Network Gateway (Gateway)	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)	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.

Pattern	Problem	Solution
Traffic Monitor (Observer)	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.

4.9 Overview

The following figure provides an overview over all patterns in our pattern language. The big colored rectangles represent the different categories. Small boxes represent the patterns, and the arrows indicate the relationships between them. The white boxes are the suggested proto-patterns, which we describe in detail in the next section. Grey boxes stand for simple, known, or adapted patterns. For a detailed presentation of these patterns, the reader is referred to our technical report [46].



5 Proto-Patterns

This section describes the listed proto-patterns from the last section in pattern-form. They have proven very helpful in the design and implementation of our own peer-to-peer system. Please note that simple, adapted or well-known patterns from this pattern language are described in detail in our technical report [46].

5.1 Application Interaction: 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. 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 of the overlay network 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.

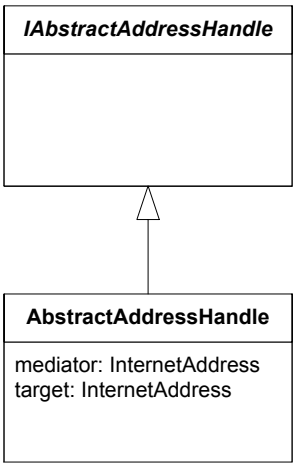
Problem

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

Forces

1. The application should be able to send a direct message to the return address that it has received from the answer of a query (Application Delivery).
2. Not all computers are well-configured in practical systems, making it impossible to simply use Internet addresses.
3. Contacting a node hosted on a badly-configured computer may need the interaction of other nodes (e.g., mediators). 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 serves as a handle to a remote node for the application. The application does not need to access the address object's internals. Therefore, Application Delivery provides an empty interface (*IAbstractAddressHandle* in the above figure) to the application, which serves as such an abstract handle. The overlay network can cast this 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 *send* method of the 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.

Rationale

Abstract Address Handles resolves most forces stated:

1. 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.

2. Because not all computers can be contacted directly, the overlay network never returns simple Internet addresses, but Abstract Address Handles.

3. 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 the address.
- 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.

Known Uses

Most 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 empty *Address* interface in FreePastry is a similar idea, even though its purpose is a little bit different in that it is not used for NAT traversal.

5.2 Messages

5.2.1 Message Hierarchy

Context

An application uses the overlay network to route an *application message* to a given key (e.g., a query message to localize an item in the network or a message to start data transfer). The overlay network itself needs to send *control messages* in order to maintain the topology. The semantics of application and control messages is very different. When an application message arrives at its target node, the Message Dispatcher has to deliver it up to the application (using Application Delivery), whereas control messages are handled within the overlay network.

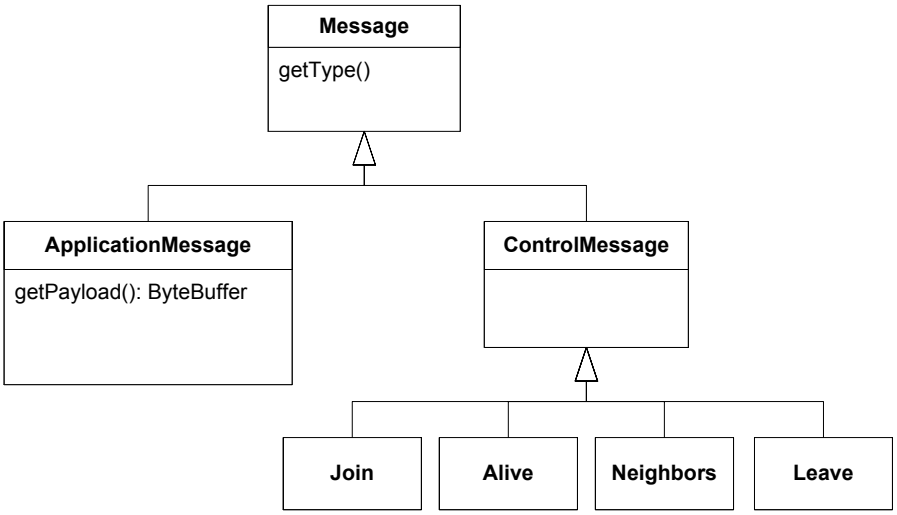
Problem

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

Forces

1. A message sent by the application cannot be interpreted by the overlay network, because it may have a different format and has a different semantics.
2. The message needs to be recognized as an application message and needs to be delivered to the application at the receiving node.
3. At least one bit of information is needed to distinguish it from other messages in the overlay network.
4. Not all messages sent by the application need to be routed, but some can be sent directly.
5. Conversely, not only application messages, but also control messages might need to be routed (e.g., join messages).
6. All control messages might share some properties and behavior.

Solution



Use Message Hierarchy to structure the messages clearly into application and control messages.

Application messages wrap messages sent by the application in a simple Envelope Wrapper. An application message can be as simple as only containing the payload. The pattern allows the Message Dispatcher to distinguish application from control messages sent by the overlay network.

Resulting Context

When marshaling, one additional bit (e.g., the type) is sufficient to distinguish application messages from control messages. When an application message is received, the Message Factory creates an *ApplicationMessage* object with the uninterpreted raw bytes as its payload, which can be delivered up to the application using Application Delivery, or the corresponding control message which is processed by the overlay network itself. Different messages can be routed by wrapping them in a Routed Message.

Rationale

Message Hierarchy resolves all forces stated.

1. The overlay network can distinguish application messages from control messages and does not need to interpret application messages.
2. Messages wrapped in an *ApplicationMessage* are delivered up to the application.
3. The information to distinguish application and control messages is encoded in the type bit of the message.
4. Not all application messages or control messages need to be routed towards their target. Instead, if they needs to be routed, they can be included in a Routed Message.
5. By making the concept of routable messages orthogonal to the Message Hierarchy (see Routed Message), all messages can be routed.
6. By having a common supertype, all control messages can share some behavior and properties easily.

The solution has the following favorable qualities:

- Clarity: Using Message Hierarchy results in a very clear structure, separating application messages from control messages in the overlay network.
- Understandability: The Message Hierarchy 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, without any changes to the overlay network.

Known Uses

Some overlay networks never send application messages directly, that is, all application messages are wrapped in Routed Messages. However, to allow the Message Dispatcher to determine whether to deliver a message to the application, this approach does not allow control messages to be wrapped in Routed Messages.

Many overlay networks do not structure the messages into application and control messages. This has the severe drawback that the design is difficult to understand and maintain.

In HyperCast [47], the overlay message header is a concrete example of this pattern [43].

References

The application message in the Message Hierarchy is simply an instance of an Envelope Wrapper. In [43], the messages are clearly separated into application and protocol messages.

5.2.2 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. The routed messages need to contain specific header fields, at least the target key of the message. If you are using Message Hierarchy, this would be a place to include those header fields. However, not all application messages need to be routed, and some control messages need to be routed as well. At intermediate nodes, all routable messages need to be treated equally. Only at their target node, different actions need to be performed.

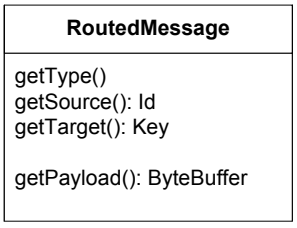
Problem

How do you make specific messages routable?

Forces

1. Whether a message is routable or not is orthogonal to other message characteristics (e.g., application messages and control messages both might be routable, or not).
2. All routable messages share some header fields, such as the target key.
3. Dependence on the message type is not favorable and can lead to code duplication.
4. Intermediate nodes should need to be able to handle the message without knowing its content or specific type.

Solution



Use a Routed Message that 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, both application and control messages can 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).

Similar to the well-known Adapter pattern, a static variant can be applied in case your programming language supports multiple implementation inheritance. In case your programming language only supports single inheritance, then 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, the Router checks whether the message is at its target or needs to be sent to the next node on the routing path.

Rationale

Routed Message resolves all forces stated:

1. By using Routed Message, both application and control messages, can be routed.
2. The common header fields are given by the Routed Message.
3. The Router and Message Dispatcher do not need to check for different types of messages that are routable. Instead, they can treat all Routed Messages the same way.
4. Intermediate nodes can treat all Routed Messages uniformly.

The solution has the following favorable qualities:

- Flexibility: Using Routed Message, it is trivial to make any kind of message routable.
- Encapsulation: The Routed Message encapsulates all header fields necessary for the routing logic.
- Clarity: Using Routed Message leads to a very clear structure. Whether a message is routed or not is orthogonal to the message hierarchy.
- 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.

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.

5.2.3 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, 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. Therefore, most systems reuse these 'universal message types'. 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 universal 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.

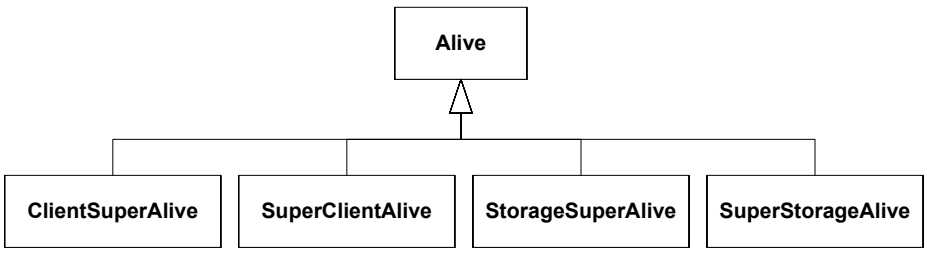
Problem

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

Forces

1. For each pair of nodes, similar messages need to be sent. Using the same message for each pair of nodes is therefore often possible.
2. For monitoring and testing, looking at the sender and receiver of a message can yield helpful information.
3. 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.
4. Using universal messages, testing and debugging the protocol is much harder because messages cannot be distinguished early, or by simply looking at their types.
5. 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.
6. Creating new message types takes some effort. They need to be added to the Message Factory, and if the marshaling and demarshaling 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.

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 effort, especially if a proprietary marshaling and demarshaling 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) is a good choice. Additionally, Source Sink Marker can be used to state this information more explicitly 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 useful 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.

Specialized Message Type resolves most forces stated:

1. 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.
2. While the sender and receiver can yield some information, it is not always possible to deduce the type of the nodes unambiguously (e.g., because different nodes are hosted on one machine, and all of them send 'alive' messages). Specialized Message Type is a simple solution for this problem.
3. In the case of virtual nodes, sender and receiver node type can be deduced when using Specialized Message Type.
4. 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.
5. Specialized Message Type also allows to contain type safe data. Instead of Node Handles, Typed Node Handles can be used for instance.
6. 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.

The solution has the following liabilities:

- Flexibility: Using universal message types is much more flexible, because no adaptations are needed when adding new node types.
- Effortless: Adding a new message type for each pair of nodes takes some effort.

- Simplicity: The message hierarchy is more complex than if there was only one universal 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

The only known application that uses Specialized Message Type is our own system. This might be due to the fact that most academic projects have only one type of node, so that Specialized Message Type cannot be applied.

5.2.4 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 checking the source and sink node type in the code at runtime at different places is not possible (e.g., in the Traffic Monitor).

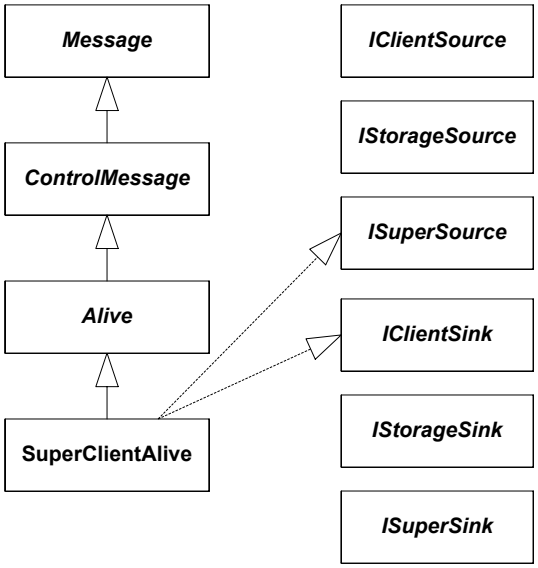
Problem

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

Forces

1. 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.
2. 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.
3. Using reflection to check the source and sink node type at runtime 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 by the program.

Create an empty source and a sink Marker Interface for each type (e.g., *IClientNodeSource*, see the above figure), and let each message implement the two corresponding interfaces. 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 bugs can be detected already at compile-time. Other bugs 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*). We provide code samples in our technical report [46, Sec. 7.2.7].

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 safer. Using Source Sink Marker, the source and sink node type of a message can be checked at runtime, which can help at different places (e.g., Message Dispatcher, Traffic Monitor).

It resolves all forces stated:

1. 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.
2. 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.
3. 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.
- 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 heavily 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, we could not find this pattern be applied in overlay networks so far.

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 [48].

5.3 Message Handling: 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 reduce 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.

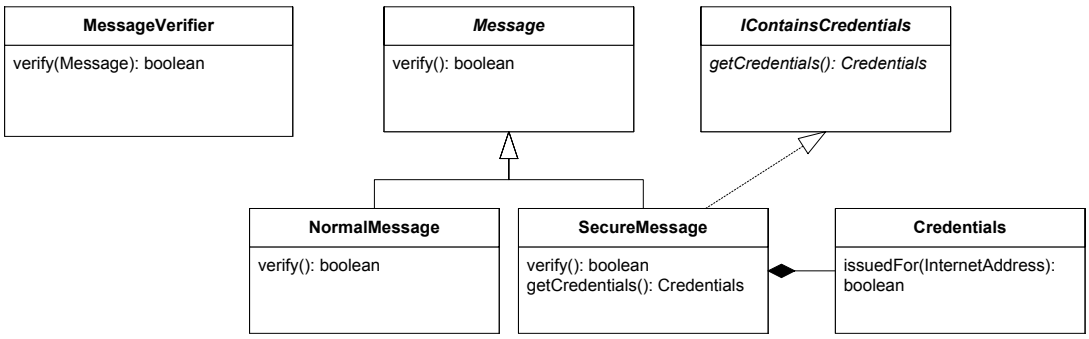
Problem

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

Forces

1. Forged messages should be detected early, before they can cause damage.
2. Some message should include credentials. It should be easy to add credentials to existing messages or to new ones.
3. Verifying the credentials is similar for all messages containing credentials.
4. The message hierarchy can be quite complex.
5. Simple verification can be done by checking header fields.
6. 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 supertype 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 one 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 (*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. We provide code samples in our technical report [46, Sec. 7.3.4].

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 object 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 superclass, 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. If multiple implementation inheritance can be used, then the verification method could also be provided by a common superclass.

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 all of the forces stated:

1. 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.
2. Not all messages need to contain credentials, because it is an overhead and makes the message larger, consuming more bandwidth. It is very easy to include credentials for any kind of message that needs it.
3. All messages containing credentials are verified the same way.
4. 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.
5. Simple verification of header fields, or applying a general policy, is very easy using the Message Verifier.
6. 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: Access and checks are encapsulated in the Message Verifier.
- Understandability: It is easy to understand when and where the 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.

5.4 Routing: 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

1. 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.

2. 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.

3. 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.

4. In some cases, the application built on top of the overlay network needs to be informed before a message is sent to the next node (using Application Notification).

5. 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.

6. Despite the complexity of the algorithm, it should be easy to read and understand where the routing takes place in the code.

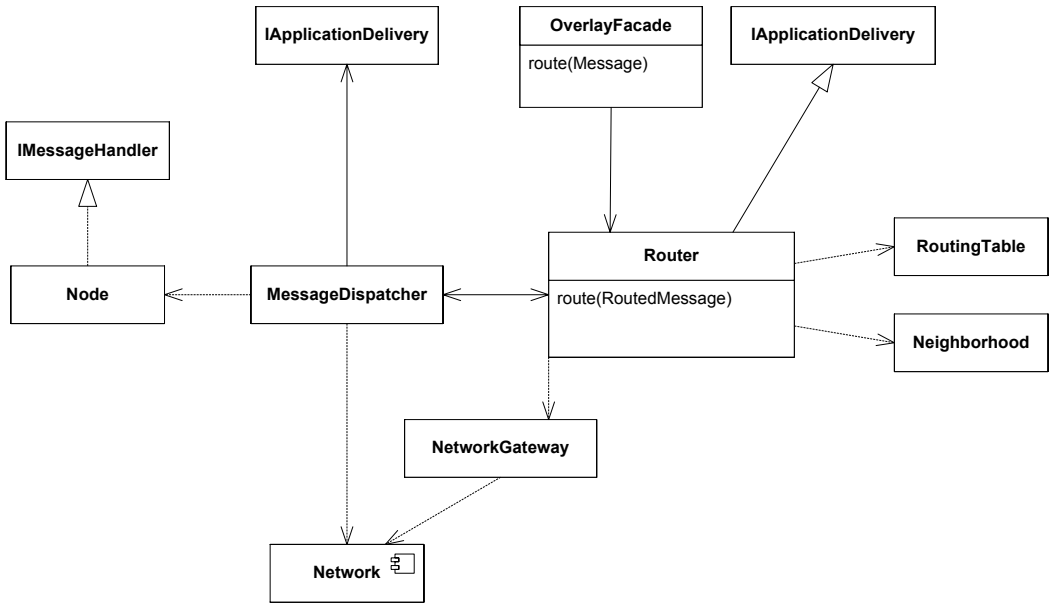
7. The same code should be used when an application uses the overlay network to route a message, and when a Routed Messages arrives at an intermediate node. This avoids code duplication.

8. 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.

9. Information about the network (Node Handles) might not only be stored in a routing table, but in a neighbor and other objects as well. This information must potentially also be taken into account in the routing algorithm.

10. Not only application messages might be contained in Routed Messages, but also control messages (e.g., join messages). Therefore, not all messages that arrive at their target need to be delivered up to the application.

Solution

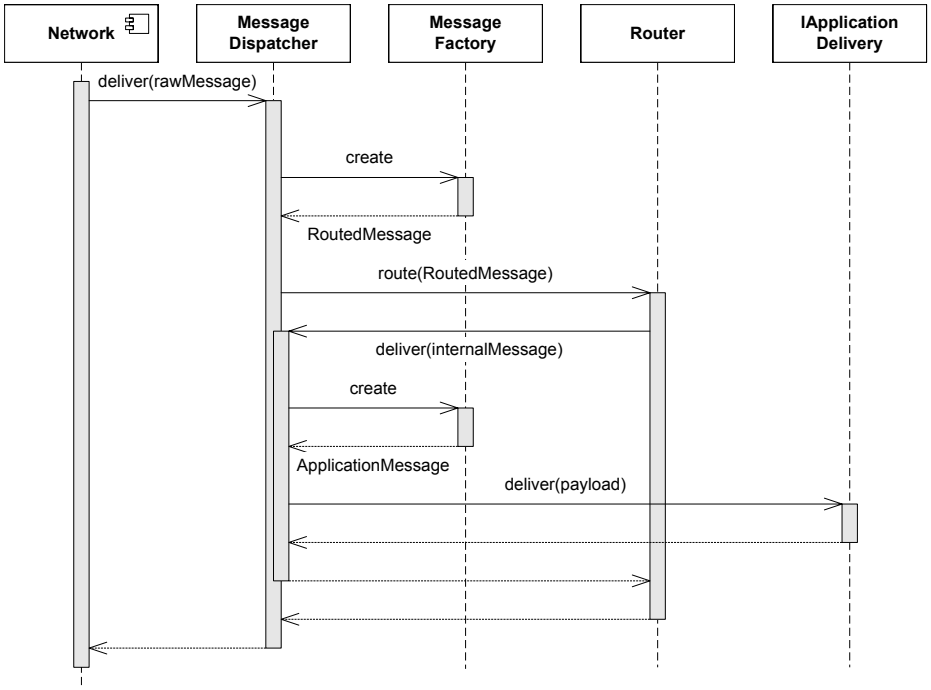


Use a Router, which encapsulates the routing algorithm and provides a method to route Routed Messages. This method checks whether a Routed Message is at its target, or if it needs to be sent away, in which case the router chooses the next node according to its routing algorithm from the routing table 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. The Message Dispatcher knows 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 is 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 application 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 an application 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., if 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. We provide code samples in our technical report [46, Sec. 7.4.1].

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 application 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 [41]. 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 all forces stated:

1. The routing algorithm is separated from the Message Dispatcher, and completely encapsulated in the Router.
2. The collaboration of the Message Dispatcher and the Router makes it possible to include the dispatching logic only once, in the Message Dispatcher.
3. Again, the collaboration of the Message Dispatcher and the Router allows to process the message in any way needed.
4. The router can inform the application built on top before it sends a message away, using Application Notification.
5. The implementation of the routing algorithm can be changed, as long as it remains semantically equivalent and conforms to the interface of the Router.
6. The Router pattern clearly localizes the routing functionality in the code.
7. 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.
8. Because of the collaboration of the Message Dispatcher and the Router, the Router does not need to know whom to dispatch which message.
9. The Router can use all information locally available, such as routing tables and neighborhood structures.

10. 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.
- 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 application messages that the application built on top wants to send.

Known Uses

A Router is used in many overlay networks, but the actual implementations vary. In FreePastry, messages arrive at the *PastryNode* (the Message Dispatcher combined with Local Node), which dispatches the Routed Messages off to the Router. The Router in turn is responsible for setting 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 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.

5.5 Protocol: 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 (e.g., its super node or its neighbor) is not alive anymore, it will need initiate a new join operation.

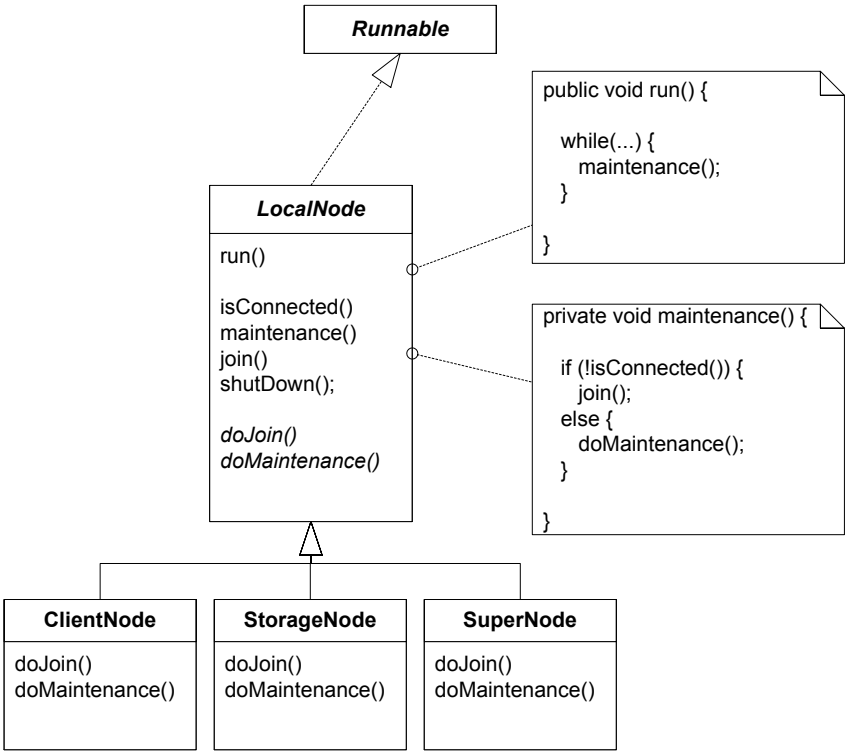
Problem

How do you implement the maintenance protocol?

Forces

- Protocol logic spread around the overlay network is hard to read and to 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 a separate thread. This makes the nodes be the only active components in the system, responsible for joining and maintaining themselves.

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 [44]). We provide code samples in our technical report [46, Sec. 7.6.1].

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 connected 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 all forces stated.

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

The solution has the following favorable qualities:

1. Encapsulation: The dynamics are encapsulated in the Local Node object.
2. High cohesion: Because everything is encapsulated in the Local Node object, this leads to high cohesion.
3. Understandability: It is easy to understand where and when each node runs its maintenance protocol.

The solution has the following liabilities:

- 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.
- Extensibility: The protocol cannot be extended easily, because the whole protocol is regarded as one block, rather than different protocols for different tasks.

Known Uses

Unfortunately, we 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.

6 Conclusions

In this paper, we presented a pattern language for overlay networks. Although many of the design issues can be addressed by simple adaptations of well-known patterns, we identified several new solutions that we suggest as proto-patterns. We implemented all of these proto-patterns in our own peer-to-peer system, a distributed storage system. This practical experience confirmed that the proto-patterns address the design issues of overlay networks effectively and lead to a clear software design.

We believe that our pattern language is helpful for both developers new to peer-to-peer systems as well as for experienced peer-to-peer programmers, who will find a collection of familiar ideas generalized to patterns.

Bibliography

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

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] Project JXTA. `http://www.jxta.org`

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

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

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

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

[15] FreePastry. http://freepastry.rice.edu/FreePastry

[16] Tapestry. http://www.cs.ucsb.edu/~ravenben/tapestry

[17] Bamboo. http://bamboo-dht.org.

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

[19] A Viceroy implementation. http://www.ece.cmu.edu/~atalmy/viceroy.

[20] LimeWire. http://www.limewire.org.

[21] jMule. http://jmule.sourceforge.net.

[22] Dijjer. http://dijjer.org.

[23] OogP2P. http://www.duke.edu/~cmz/p2p.

[24] GISP. http://gisp.jxta.org.

[25] Azureus. http://azureus.sourceforge.net

[26] JTorrent. http://sourceforge.net/projects/jtorrent.

[27] Meteor. http://meteor.jxta.org.

[28] 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.

[29] Skype. http://www.skype.com.

[30] Jabber. http://www.jabber.org

[31] Gnutella. http://www.gnutella.org

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

[33] eMule. http://www.emule.org.

[34] 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.

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

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

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

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

[39] OpenDHT. http://www.opendht.org.

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

[41] 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.

[42] 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.

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

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

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

[46] D. Grolimund. Design Patterns in Peer-to-Peer Systems. A Pattern Language for Overlay Networks. Technical Report 503. ETH Zurich.

[47] HyperCast. http://www.cs.virginia.edu/~mngroup/hypercast.

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