1 Introduction

Despite of the popularity of message-oriented middleware (MOM) in recent years, little has been achieved in terms of interoperability of application and middleware components that have not been designed to operate together in the first place. This is partly because standards tend to provide flexible infrastructure filled with life on application level without committing themselves to common usage scenarios (like older AMQP versions [1]), because of schisms in protocol communities (like in the AMQP community [1, 10]) and because of standards that contain themselves completely from any commitments concerning middleware behaviour (like Stomp [2]). In consequence, application developers invent their own interoperability scenarios, rely on specific interpretations of standards or, and even more so, rely on proprietary features of the brokers used in their project. Changing brokers, not to mention protocols, or integrating components developed in other contexts is usually not possible without significant code changes.

We, therefore, present a set of communication patterns that describe common ways of applications to interoperate with each other. Patterns are graphs consisting of applications (vertices) and queues (edges) with additional structural and behavioural characteristics enforced by APIs and other middleware means. The aim is to build a framework of patterns that provide interoperability means with common behaviour and with common reliability characteristics on top of available middleware, but independant of any specific middleware and, in particular, independant of advanced features. We assume the simplest middleware possible as a common reference point that can be used to implement patterns. The only feature expected from middleware is some kind of *queue* device that allows applications to write messages to it on one end and to read these messages from the other end. On top of this device, we build patterns like *client/server*, *publish and subscribe*, *pipeline*, *router*, *forwarder* and so on.

We are aware that this is not the first paper starting with moaning about limitations of available interoperability frameworks and we do not expect to have invented a way to overcome the issue once and for all. But we hope to contribute to a middleware and application design that, together with other initiatives, standards and implementations, improves interoperability of application and middleware components.

The notion of pattern, as it is presented here, borrows heavily from Pieter Hintjens *Code Connected* [5] and the *ZeroMQ* community. The patterns defined by Hintjens are designed for a middleware library and part of the very core of this library. Patterns here aim at independence from specific middleware and, as such, aim at reusability of applications among middleware and protocols.

With *ZeroMQ* as source of inspiration, it cannot be denied that the pre-1.0 versions of AMQP [1], as well, have influence on the present work. There are however some important differences. The AMQP specification defines the infrastructure that can be used to create specific interoperability means, but it does not define those means. The AMQP prefers abstract principles over concrete use cases. The infrastructure underlying our patterns is much simpler than the middleware defined by AMQP. Many of the ideas related to AMQP, such as the *chain of responsibility* for MOM proposed, for instance, by Steve Vinoski [11, 12], have been simplified beyond recognition; as in the *ZeroMQ* context, we prefer concrete use cases over general principles, *e.g.* a service broker, instead of a general crystal glass for finding the handler for a given message. If something as flexible as this is needed in practice, it can be built based on application-specific rules much more simply than based on general patterns.

From far, there is of course some influence from *Design Patterns* by the *Gang of Four* [4]. Our approach has reusability as one of the goals in common with the *GoF*, but deviates in many respects. The *GoF* patterns are about code design, whereas our patterns describe structures of interoperability not related to code at all.

The paper is organised as follows. Section 2 describes the model that we use to describe patterns and provides a set of definitions of basic terms. In section 3, we present a set of basic and advanced patterns that are directly related to interoperability. We will not discuss patterns that aim at specific services like message storage or distributed shared memory. In section 4, we present an implementation in Haskell assuming Stomp as underlying middleware. The rationale for using Stomp is to demonstrate the patterns with one of the simplest protocols available. The reasons for using Haskell are outside the scope of this paper. Secion 5 provides some conclusions.

2 Model

To model queuing patterns, we will assume two finite sets, a set $A = \{a_1, a_2, ...a_n\}$ of applications and a set $Q = \{q_1, q_2, ...q_n\}$ of queues.

Applications, *apps* for short, are units of computation that may consume input from a queue and may produce output into a queue. The type of computation going on within an application is usually not considered in this context and irrelevant for us. In some cases, however, the processing can be completely described in terms of input and output observable in queues. These applications are essential parts of queuing patterns (*e.g.* router, forwarder, bridge) and will be explicitly named as such. Important is that applications do not share any resources besides queues. Interoperability between applications, hence, is entirely through queues.

By means of input and output of queues, we can distinguish different types of computation perfomed by an application. In this definition, we take only input and output into account that is relevant within the subset of applications and queues that are currenlty under consideration. Within a pattern, an application may take input produced by another application and use it to generate output that is directed to that application. We call this type of computation a *service*, an application performing such computation a *server* and a consumer of a service is called a *client*. Note, however, that we sometimes use the term *service* to refer to its common meaning. This should be obvious from the context of its usage and we will not emphasise one or the other meaning by special typesetting rules.

An application may then take input, but not provide any output. This type of computation is called a *task*, the performing application is called a *worker* and the requesting application a *pusher*. Finally, an application may produce output without any input requesting this output. This type of computation is named a *topic*, the app providing a topic is called a *publisher* and the app receiving topics is the *subscriber*. The following table summarises the three types of computations:

computation	input	output
service	X	X
task	X	
topic		X

There is an important difference between patterns involving services and patterns involving tasks or topics. Tasks and topics do not provide feedback to their consumers, a pusher, for instance, would request a worker to perform a task, but would never know whether the task has been performed successfully or not, since the protocol does not foresee such a status report. Subscribers, on the other hand, receive data from a publisher periodically or sporadically. If the topic is provided periodically, the subscriber is able to conjecture whether the publisher is working properly or not within given time periods. Whether the subscriber can rely on the publisher according to its own reliability requirements, depends strongly on the intervals between updates. For sporadic publishers, this does not hold. The subscriber will know that the publisher is working properly only when a new dataset has arrived and will gradually return to incertainty with time elapsing after the update. A common strategy to improve reliability for tasks and topics is therefore the use of heartbeats, *i.e.* messages that the provider (publisher, worker) sends periodically to consumers (subscriber, pusher) without a special meaning besides the fact that data have been sent from that source. The interval between heartbeat messages depends on the reliability requirements of the consumer.

Clients do not need heartbeats from servers, because the protocol implies a response, *i.e.*the availability of the server is revealed in the moment of its usage. Clients, however, should use timeouts to avoid waiting eternally on the expected response. The timeout interval depends on the time needed by the system (server and interoperability framework) and the reliability or time requirements imposed on the client.

Queues are best defined according to Knuth [9], p. 239:

> A *queue* is a linear list for which all insertions are made at one end of the list; all deletions (and [...] all accesses) are made at the other end.

We call the end at which elements are inserted the *tail* of the queue and the end from which elements are read or removed the *head* of the queue. It is not possible to read or delete a message that is currently not at the head at the queue, *i.e.* we can only read the oldest element in the queue. Additionally, we cannot insert an element anywhere but at the tail. It is this way guaranteed that elements in a queue are always read in the order of their insertion.

We assume that there is some kind of API enabling applications to write to queues (*enqueue*) and to read from queues; reading always implies deletion (*dequeue*), it is thus not possible to get access to an element in the queue without removing it.

The elements in the queue are called *messages*. Messages are chunks of data, it is up to applications' internal structure, which, as we said, is irrelevant for our discussion on queuing patterns, to make sense out of those chunks. We assume, however, that there is a protocol envelope around messages that may carry information directed to the middleware and, hence, relevant to queuing patterns. This information may have the form of *headers* in the sense of Stomp, for instance, but other formats are possible, *e.g.* message segments preceding the core message as in the protocols built on top of *Zeromq* [5].

We do not assume any specific context, processing platform or programming language to host queues. Queues may be part of a processing context like Unix processes and connect threads running within the same memory space; queues may also be part of an operating system in the form of a socket library – UDP would be a promising candidate for such a solution, but also advanced socket concepts such as *Zeromq* may be considered; queues can also be hosted on a broker to which potentially remote apps connect through sockets. Nothing is assumed that would prevent any of these scenarios. In particular, no assumptions are made about advanced queuing features like transactions, receipts, multi-segment messages and so on.

Unsurprisingly and already implied by the previous paragraphs, applications enqueue and dequeue messages to and from queues. We describe this formally by means of two relations, $P \subseteq A \times Q = \{(a_i, q_j), \dots\}$ and $C \subseteq Q \times A = \{(q_i, a_j), \dots\}$. The relation P describes data production, *i.e.* for a tuple (a_1, q_1) in this relation, a_1 enqueues messages to q_1; the relation C describes data consumption, *i.e.* for a tuple (q_1, a_1), a_1 dequeues messages from q_1. Obviously, we can create chains of enqueue and dequeue operations by creating lists of tuples from P and C:

$$[(a_1, q_1), (q_1, a_2), (a_2, q_2), (q_2, a_3)] \tag{1}$$

$$[(a_1, q_1), (a_1, q_2), (q_1, a_2), (q_2, a_3)]. \tag{2}$$

We may shorten structures shown in the first example by using the symbols $\rightarrow$ (enqueue) and $\Rightarrow$ (dequeue): $a_1 \rightarrow q_1 \Rightarrow a_2 \rightarrow q_2 \Rightarrow a_3$ and we use the symbols $\triangleleft$ (enqueue to all) and $\diamond$ (enqueue to one) to represent structures shown in the second example: $[a_1 \triangleleft [q_1, q_2], q_1 \Rightarrow a_2, q_2 \Rightarrow a_3]$ or: $[a_1 \diamond [q_1, q_2], q_1 \Rightarrow a_2, q_2 \Rightarrow a_3]$.

By using lists for both kinds of chains, we imply ordering, not necessarily temporal ordering, but logic ordering for sure. In the first example, we must perform the operation $a_1 \rightarrow q_1$, before the operation $q_1 \Rightarrow a_2$ has any effect. The same is true for operations $a_1 \triangleleft [q_1, q_2]$ or $a_1 \diamond [q_1, q_2]$ and and $q_1 \Rightarrow a_2$ in the second example. If any time elapses between the first operations and the effect of the second ones is irrelevant, besides some remarks on performance we may make in the following sections.

The operations $\triangleleft$ and $\diamond$ are multiple enqueues. The multiple enqueue to all, $\triangleleft$, has the same effect on all queues to its right side as the simple $\rightarrow$ to the single queue on its right side. The effect of the operation $a_1 \diamond [q_1, a_2]$ is different. It is in this case the first queue only on which the operation has effect. The state of the following queues is unchanged. The ordering of the queues following $\diamond$ is therefore significant. The operation, however, has yet another effect: the head of the list of queues $[q_1, q_2]$ is removed and added to the end of the list. The result is thus: $[q_2, q_1]$ and the next application of the operation will favour q_2. Note, however, that this behaviour is not expected to be supported by the underlying queuing middleware. It has to be implemented by patterns as described in the following sections.

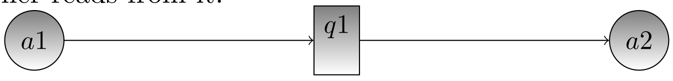
To ease reading, we introduce a graphical notation. We represent applications by nodes of the form $a1$ and queues in the form of $\boxed{q1}$ where queues may have attributes like: $\boxed{\begin{array}{c} q1 \\ \text{channel} \end{array}}$. This would mean that $q1$, as part of its protocol envelope, has an attribute named *channel*. Enqueue and dequeue operations are uniformly depicted as arrows either pointing to a queue (enqueue) or pointing to an app (dequeue). Arrows may also connect lists of apps and queues that are named as $a_{i,\dots,n}$ and $q_{i,\dots,n}$ respectively. A simple arrow connecting such a list of apps with a list of queues should be interpreted as one enqueue operation per pair of app and queue, *i.e.* $a_i \rightarrow q_i, a_{i+1} \rightarrow q_{i+1}, \dots, a_n \rightarrow q_n$. When a multiple enqueue is intended, the corresponding arrow is marked with an operation symbol, either $\lhd$ or $\diamond$.

3 Patterns

3.1 Basic Patterns

3.1.1 one-to-one

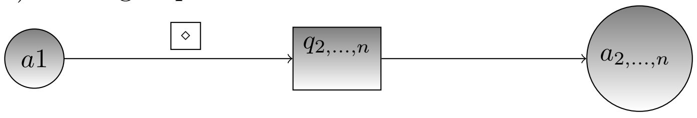
The simplest possible pattern in our model is a pair of two applications and one queue, whose name is known to both applications in advance. One app writes to the queue, the other reads from it:



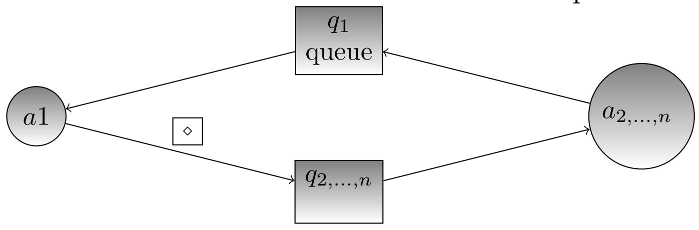
This one-to-one pattern describes the minimal requirement for any broker or other middleware underlying our pattern library. Even if there are certainly many use cases for this structure, it appears half a pattern or, halves of two different patterns joint together. According to the definitions above, *app*1 appears to be a publisher publishing data under some topic, *app*2, however, appears to be a worker, receiving a request to perform some task. Both is possible at the same time. For this general purpose structure, the function that is fulfilled in concrete depends entirely on the application context.

3.1.2 pipeline

A worker, in the strict sense, is realised with a pattern, sometimes called a pipeline [5]. A pipeline consists of set of applications ($a_{2,\dots,n}$) receiving requests and an application ($a1$) sending requests to one of the workers:



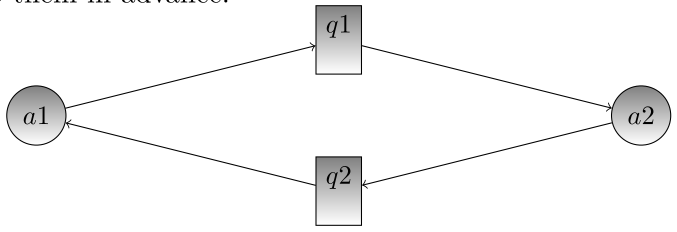
Since operation $\diamond$ is used in the pipeline, tasks are assigned to the workers in round robin fashion. The pipeline, hence, is an example of a simple work balancer. It is noteworthy, here, that the round robin logic is implemented in the sender application. Furthermore, according to the diagram above, the *client*, *i.e.* the application pushing the task, knows the available workers and how they are reached through queues beforehand. Since this is obviously a quite static configuration, we should extend the pipeline to a scheme where workers can connect to the pusher:



The pusher would now wait for workers to register through queue $q1$ with the name of the queue through which they expect task requests to be sent and then start to send its task to available workers. But now, an issue arises that will occur in all situations where an application registers at another. The workers perform a task on behalf of app $a1$, but since workers give no feedback about the results of the task, there is no way for $a1$ to know whether the task has been done. The application even does not know if the workers are still available after some time, they – or at least some of them – could have disconnected silently. Since application $a1$ depends on the workers, there should be a mechanism for $a1$ to know that the workers are still available. We will therefore introduce end-to-end heartbeats. Part of the registration protocol is a negotiation step, during which the pusher and the registering worker decide whether in at which rate the worker should send heartbeats through $q1$. If heartbeats are missing for some turns, including a tolerance phase, the pusher should automatically deregister the worker in question. It is not necessary that the pusher sends heartbeats in return, since the workers do not rely on the pusher.

3.1.3 peer-to-peer

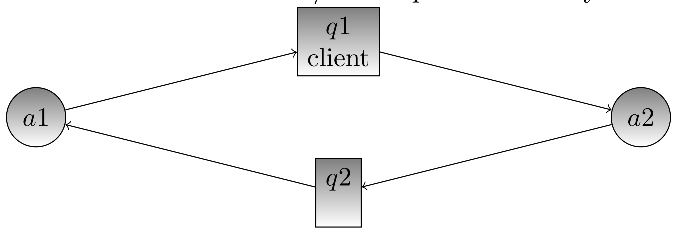
The peer-to-peer pattern is again a general-purpose structure that is intended for building higher level patterns, like, for instance, the *exclusive star* [5]. The pattern consists of two applications that communicate with each other through two queues that are known to them in advance:



There are no further rules imposed on the communication of the two peers and no heartbeats are exchanged. All further mechanisms are defined by the application or high-level pattern using this structure.

3.1.4 client/server

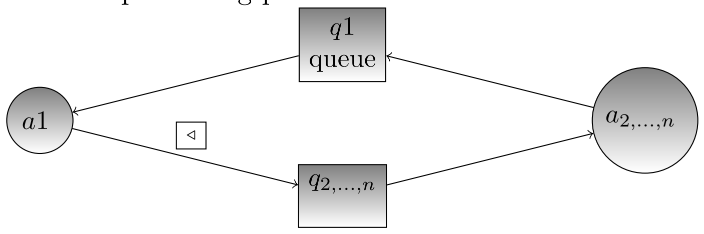
The structure of the client/server pattern is very similar to the peer-to-peer pattern:



A difference is a mandatory attribute in the protocol envelope of a request sent by the client, $a1$, to the server $a2$, the *client* header. This header contains the name of the queue through which the client expects the reply. This implies another restrictions on the communication: The client must send a request to the server, before the server can do any work. The reply will then be sent through a queue named according to the *client* header, $q2$ in the diagram above.

3.1.5 one-to-many (publish/subscribe)

The publish/subscribe pattern is similar to the pipeline in that there is a registration part and a publishing part:



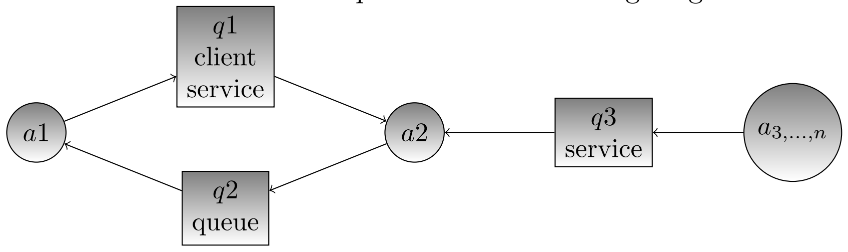
The subscribers on the right-hand side of the diagram ($a_{2,\dots,n}$) would register through queue $q1$ indicating a the name of the queue through which they will expect the published data. The publisher will send its data to all queues in its registry indicating the topic. Since, in this case and this is an important difference to the pipeline model, it is the receiver, *i.e.* the subscriber, that relies on the data sent by the publisher, it should be the publisher to send heartbeats to the subscriber. If heartbeats are missing, the subscriber can decide to connect to another publisher for this topic available in the system.

3.2 Derived Patterns

Derived patterns are advanced concepts using basic patterns to implement more specific solutions. The number of possible and useful patterns is endless. We will focus, here, on a small set of patterns that primarily aim to extend the possibilities of basic patterns in connecting applications with each other and to improve reliability.

3.2.1 Service desk

A common issue with messaging software is how to manage all the access points applications have to know of each other. With local configurations providing such information to each node scattered in a huge network makes reconfiguration a hideous task. We want as little information about single access points including host names, ports, configuration parameters or simply queue names distributed to applications. A simple solution for this problem is an information service similar to yellow pages. The client application would send a request to this server asking for the address of another service, a task or a topic and would receive in reply the name of the queue through which that service would be delivered. Such a service is depicted in the following diagram:



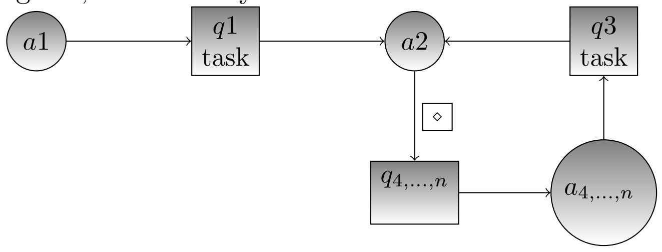
Here, $a1$ is the client application sending a request through $q1$ to the server ($a2$) that consists, as for all client/server configurations, of the reply queue of the client (*client*) and the enquired service (*service*). Service, task or topic providers ($a_{3,\ldots,n}$) register at the server under their own service, task or topic name. If more than one provider is registered for a service, $a2$ will register a list of providers for this service and round-robin through this list. The head of this list is send back through the client's reply queue. If no provider of the service in question is available, the service desk should respond with a message like NOT FOUND.

To improve the service desk's reliability, the providers can send heartbeats to the service desk. If a provider fails to send heartbeats for some turns, $a2$ would remove this provider from the list for this service, task or topic. The service desk itself may also provide heartbeats to providers. If the service desk fails to send heartbeats, the providers should switch to an alternative service desk.

3.2.2 Task load balancer

In many situations, the service desk is a good solution, in particular for topics, to which the client would connect only once and receive the topic data continuously, or for services and tasks that are rarely used or used only once throughout the lifetime of the client application. In situations, however, where the client needs to request a task or service repeatedly, the service desk may be uncomfortable. Many things may happen in the world between the request of the service and one of its subsequent uses. The service, hence, may not be available anymore, when the client tries to invoke it. The client either accepts this disadvantage, requesting the access point at the beginning, using it until it fails and then requesting another one, or it has to refresh it periodically or even each time when it is invoked. The last option would double the number of requests effectively performed to receive the same service, whereas the previous imposes some management overhead for access point information without a safe guarantee that the service at this access point is always available.

A better solution for this cases would be not to provide the access point to the client, but to perform the task on its behalf right away. This solution also has the side effect that requests can be balanced just in time. The task work balancer, shown in the following diagram, does exactly this for tasks:

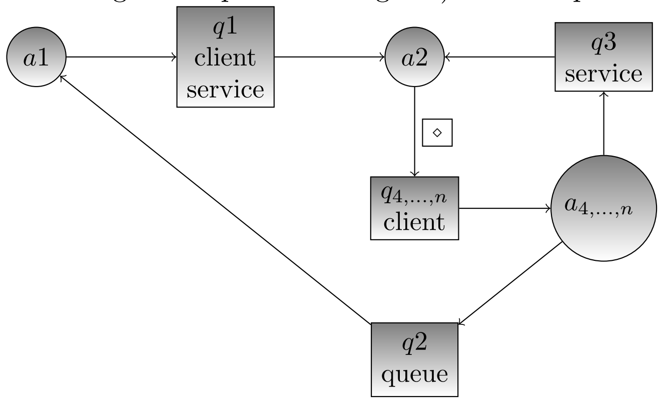


Just like the service desk, workers register at the balancer ($a2$) through queue $q3$ indicating the task name. The balancer, on receiving a client requests through $q1$, forwards the request to on of the queues $q4, \dots, n$. Again balancer and workers should exchange heartbeats to improve the reliability of the service.

From the client's perspective, there still remains an issue. The task balancer, basically, shifts the initial problem, the possibilities of tasks failing, one level up, *viz.* to the balancer itself. There is no ultimate solution to this issue. The balancer should belong to the basis of trusted middleware and, hence, must be implemented with great care to avoid faults caused by software errors. If this guarantee is insufficient, the client application has to rely on itself, requesting a redundant set of balancers right from the beginning and switching to the second one, when the first fails, to the third, when the second fails and to the k^{th} when the $k-1^{st}$ fails. The issue, however, is not limited to the balancer; it also extends to the whole middleware, in particular when a broker is used. For clients with demanding safety requirements, redundant configurations for the entire middleware and all services should be used.

3.2.3 Service load balancer (Majordomo)

A service balancer can be built in a similar way, one more component is needed. Since services include a reply, the client's reply queue ($q2$, which, as the reader may have noted, was missing in the previous diagram) must be passed through to the final server:

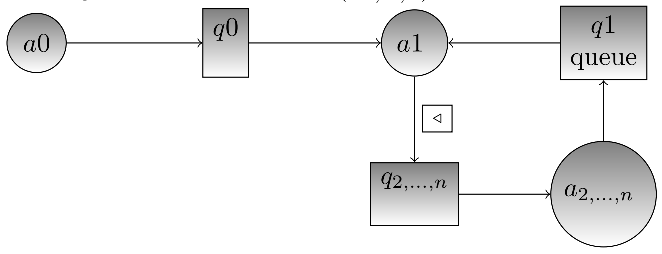


The service load balancer ($a2$) receives a request including a *client* header. This header is passed through to the servers ($a4, \dots, n$) and the servers send the reply back to the client using the indicated queue.

The service load balancer is analogue to the majordomo protocol (MDP) in *Zeromq* [5, 7]. A difference is that, in the MDP, it is the balancer – or broker, as it is called there – to relay the reply back to the client. Here it is assumed that the backward channel can be passed as an attribute to the server and that the server is able to use this channel to send the reply autonomously.

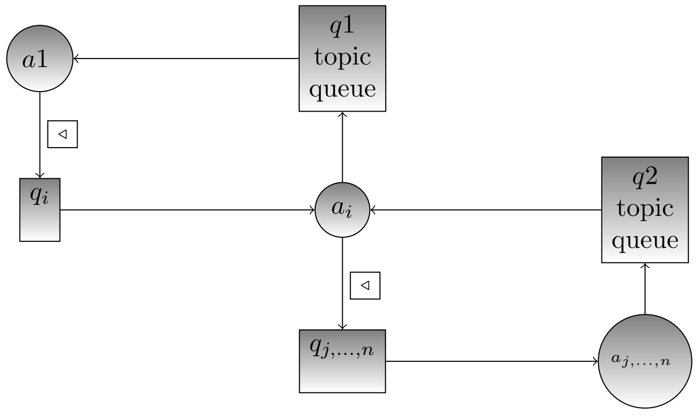
3.2.4 Router

Above, in section 3.1.5, we have introduced a publisher that creates some data and sends it to a set of registered subscribers. The router is a slight variation of that pattern. The router is basically a publisher that uses a subscriber to *create* its data. It, hence, allows separating the two concerns of creating data and publishing them, it provides publishing as a service. The original data creator ($a0$) would enqueue messages to just one queue ($q0$), from where they are read by the router ($a1$). The router, as an ordinary publisher, provides an interface to register topics ($q1$). Data received through $q0$ are then routed to all registered subscribers ($a_{2,\ldots,n}$):



3.2.5 Forwarder

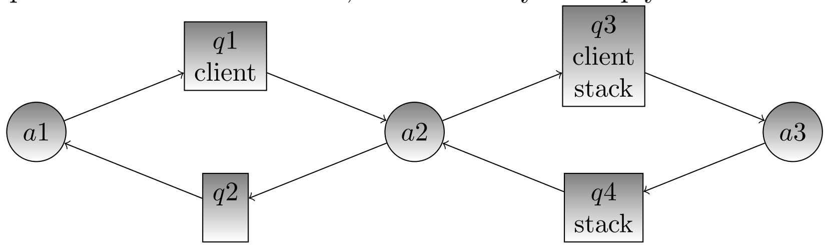
The forwarder is an extension of the router pattern. Application $a1$ publishes data to set of subscribers (a_i). The single-indexed queue and application (q_i and a_i), here, represent one of a set of queues and applications, *i.e.* the applications but the i^{th} one are not considered here, they may be regular subscriber applications. a_i, however, is again a router that forwards the received data to a set of registered applications ($a_{j,\ldots,n}$):



This way, the forwarder prolongs the reach of the initial publisher. This is in particular interesting, when the application $a_{j,\dots,n}$ are located on another broker. A forwarder, hence, is a way to bridge publishers from one broker to the other. To provide this function, a_i needs to be connected to both brokers, reading from q_i on the first and reading from q_2 and writing to $q_{j,\dots,n}$ on the second.

3.2.6 Service bridge

This is not possible in this simple way for pairs of clients and servers. Again, clients need feedback from server and, therefore, the feedback must be relayed back to the client. If client and server are indeed hosted on different brokers, it is not sufficient, as it is for the load balancer, just to pass the reply queue to the server. The reply queue, in this case, does not exist on the server's broker. The bridge, hence, must not only pass the request forward to the server, but also relay the reply back to the client:



The application $a2$ in the diagram represents the bridge connected to both broker. It acts as a regular server to the client and, on the other side, it acts as client, forwarding the request to $a3$. It sends the request through $q3$, where its own reply queue existing on the same broker as the server $a3$ is indicated in the *client* header. In the *stack* header, it passes the reply queue of the initial client $a1$, which sits on the first broker. The server $a3$ should should pass the *stack* header back to the bridge. The bridge would then use this attribute to find the queue to which this specific reply should be written on the other broker. The header is called *stack* because it may carry many queue names. This way, the pattern is exendable, *i.e.* more than on bridge can be linked together to connect one broker with the other. Each bridge would then read the top of the *stack* and write the reply to the corresponding queue popping the first element from the *stack*. The whole way to the initial client can thus be traced back easily.

3.2.7 Other patterns

The patterns presented above are pieces of an infrastructure framework to connect applications with each other in a meaningfull way. The patterns impose roles on applications that they have to fulfil in communication with other applications, such as being a client, server, publisher or subcriber. We have not yet discussed patterns that add value beyond interoperability as such. Most patterns proposed in literature, of course, do implement services beyond interoperability, it is these services in the end, for which interoperabity is usefull in the first place.

An obvious pattern in this sense is messages persistence and recovery. For persistence, different motivations and, hence, different implementations are possible. One intention is reliability even in case of a broker failure. Messages not delivered at broker failure would be lost and in many cases it would be difficult to tell the exact state of sender, broker and receiver relative to each other, so it would be difficult, if not impossible, to restart at the state where the system was interrupted by the failure. A persistence service would engrave messages into some medium before they are sent to the final receiver. Many scenarios are possible to use such a message repository. The messages in the repository could be marked as handled on receipt of an acknowledgement from the receiver and then be removed later. In such a scenario, the receiver would be responsible for the message life cycle. An alternative approach is the Titanic pattern [5, 8], used in the context of *Zeromq*. The Titanic pattern is designed client/server communication. The client registers a request at the Titanic server. The Titanic server engraves the request and relays the request on behalf of the client to the final server. The server reply is again engraved as part of the request dataset. The client would later request the current state from Titanic, which, then, would return the result. In this case, it is the client that controls the life cycle.

Part of services to save messages to some external medium is of course recovery. Receiver may need *persistent queue*, *i.e.* queues that maintain messages, when the receiver is not connected to the queue. In this case, the recovery process would start, when the receiver connects and, with this, requests the messages received in in its absence. Other scenarios may involve more complex recovery scenarios where interoperating applications first establish the current state, for instance, after a crash of parts of system.

Another useful pattern is a configuration manager. To ease configuration, applications would request configurations from a central registry server that provides the current version of this application. Such a service could use a timestamp to compare configuration available at server and client side. Is the server's configuration newer than that of the client, the new version would be deployed; otherwise, the client continues to work with the configuration already available on client side. More complex scenarios using distributed memory to store configuration variables for all applications would be solution supporting just-in-time configuration changes.

Distributed memory, like distributed search trees, would be an interesting pattern in itself. Single applications would implement parts of that tree passing search request around until a match is found or a termination condition is met. In the *Zeromq* universe, such a protocol indeed exists, the *ZeroMQ Clustered Hashmap Protocol* [6].

For big data in general, a MapReduce protocol [3] would as well be an interesting pattern. The algorithm would start by sending tasks to a set of workers on different servers in the cluster; The workers would then forward their results to the reducers, which, in their turn, would deliver the final result to the client that, initially, had started the process.

Patterns, as can be seen, move up the abstraction chain coming closer and closer to application-specific tasks. There is thus not a final and definite set of patterns, which should be implemented in all frameworks, but there should be, instead, general-purpose frameworks implementing fundamental communcation patterns as described above and more specialist frameworks for different levels of quality of services (*e.g.* high reliability patterns) and different application domains. Higher level libraries would come very close to service-oriented architecture frameworks providing not only gernal-purpose task, but also domain-specific ontologies and business logic processor engines.

4 Implementation

5 Conclusions

Literatur

[1] AMQP Working Group. Advanced Message Queuing Protocol (AMQP), version 0.9.1. `http://www.amqp.org/specification/0-9-1/amqp-org-download`, 2008. [retrieved 10 September 2013].

[2] Codehouse. Simple Text Oriented Message Protocol, version 1.2. `http://stomp.github.io/stomp-specification-1.2.html`, 2012. [retrieved 10 September 2013].

[3] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, 2004.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented Software.* Addison Wesley, 1994.

[5] Pieter Hintjens. *Code Connected Volume 1 - Learning ZeroMQ, Professional Edition for C/C++.* O'Reilly, 2013.

[6] iMatix Corporation. CHP - ZeroMQ Clustered Hashmap Protocol - Draft. `http://rfc.zeromq.org/spec:12`, 2011. [retrieved 10 September 2013].

[7] iMatix Corporation. MDP - Majordomo Protocol. `http://rfc.zeromq.org/spec:7`, 2011. [retrieved 10 September 2013].

[8] iMatix Corporation. TSP - Titanic Service Protocol. `http://rfc.zeromq.org/spec:9`, 2011. [retrieved 10 September 2013].

[9] Donald Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Addison-Wesley, 1997.

[10] OASIS. Advanced Message Queuing Protocol (AMQP), version 1.0. `http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf`, 2011. [retrieved 10 September 2013].

[11] Steve Vinoski. Chain of responsibility. *IEEE Internet Computing*, 6:80–83, 6 2002.

[12] Steve Vinoski. Advanced Message Queuing Protocol. *IEEE Internet Computing*, 10:87–89, 10 2006.