

# **Application scenarios**

Version 1.2

Written by Michael Wittmann

# Table of Contents

Chapter 1: The Change in the paradigm .....	3
1.1 From Client-Server .....	3
1.2 ... to p2p networks .....	6
1.3 ... to Space-based data storage .....	7
1.4 „Being more honest“ .....	10
Chapter 2: Usage Possibilities for XVSM.....	12
2.1 Adding more computation power.....	12
2.2 High-priority requests .....	14
2.3 Execution path .....	15
2.4 Execution path with load balancing .....	19
2.5 Execution path with automatic load balancing .....	22
2.6 Execution path with recovery after failure .....	28
Conclusion.....	31

# Chapter 1: The Change in the paradigm

## 1.1 From Client-Server ...

When designing software, the designer must decide what architecture to use, especially if the requirement exists to distribute the data over multiple computers. There are abundant possibilities to design the software in a way that fulfils this requirement. One of the most common techniques is the **Client–Server architecture**. In this architecture, there is one server which holds the information. On the other hand, there is a (possibly big) number of clients that need information that the server has, so they contact the server to retrieve that information (see fig. 1.1).

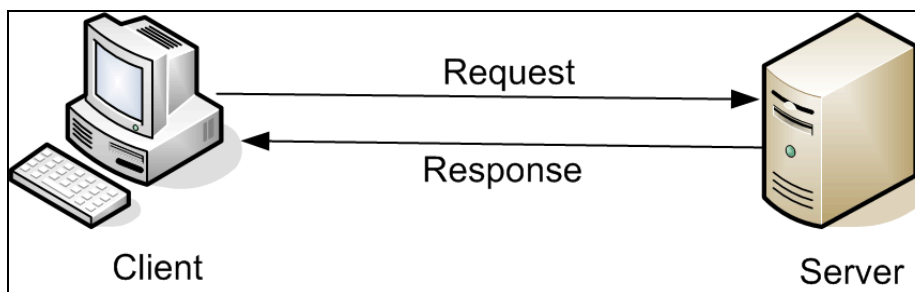


Figure 1.1: Client-Server architecture

This leads to some problems, which are already well-known by most developers, who are using the Client–Server architecture: when the server **crashes**, the information offered by the server is not available any longer. It is even worse as the server typically doesn't just offer data services but it offers more sophisticated services to the clients. So if the server crashes, the complete service is unavailable. Imagine a factory where the server is the central point that sends commands to the production machinery. If the server crashes, all machines would possibly stand still which could lead to heavy losses for the company.

Another situation is that the centralized server does not crash, but is unavailable. For example, it might happen that a server is heavily **overloaded** with requests. The server needs some time to receive a request, parse it, retrieve the information and send the answer back. In many cases, the answer is given back within (milli-) seconds. With a big number of requests, the server might need many seconds or

even minutes until it has processed all requests until being able to answer to the most recently sent request (see fig 1.2).

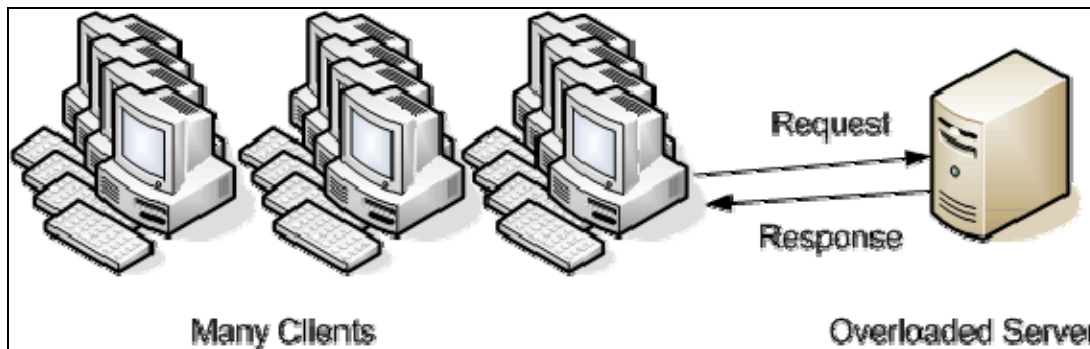


Figure 1.2: An overloaded server

In some networks, especially in the Internet, unreliable communication is used, which means that it is not sure that information is really delivered to the server or back to the client – it could be lost on its way. Thus, there are timeouts used, which give an interval in which the other communication partner must give an answer, else it is assumed that the information is lost. Talking about overloaded servers, which take seconds or minutes to give an answer to a request, this could mean that the server is up and running, still processes an answer, and meanwhile the client **times out**. Normally, the request is sent repeatedly, until the client gives up and decides that the server is down. But resending a request multiple times, that is also processed multiple times, could lead to problems if the request is not idempotent; this means it changes the state of the server on every call. For example, imagine you'd like to order some goods. If the request is sent multiple times to the server, which processes this order, but does not respond within the timeout, the goods may be ordered multiple times. Special treatment for such situations must be implemented.

As a consequence, many companies invest in a **set of servers** to service the clients. There are different approaches how to use a set of servers, like having one server and if it fails, to let the backup server fulfil the requests. Or to equally distribute (“load balance”) the requests to all servers to have a good utilization per server and thus a low response time for each request (see fig 1.3)

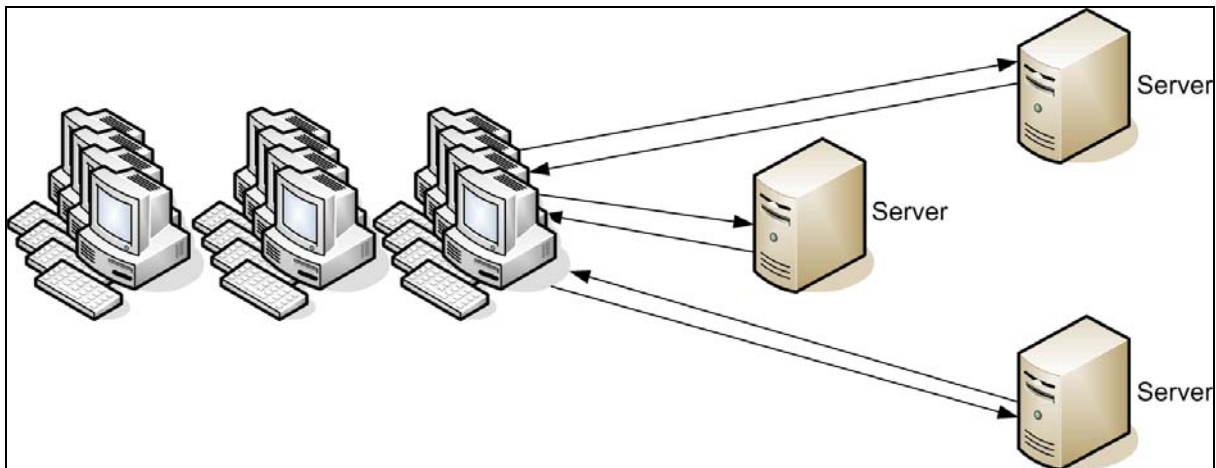


Figure 1.3: Multiple servers answering to requests

In most cases, the servers must react the same way when talking with the clients. The clients are not interested in getting different responses when communicating with different servers. Imagine two servers where one server has one part of the database and the other server has the other one, whereas the requests are handed to a server randomly. In this scenario - depending on to which server the request is routed to - one time the request can be fulfilled and the next time, the information might not be available, as it is standing in the other part of the database. So for this scenario, it is required that the servers must have access to the same information. In many cases, a shared database is used. In this case, the bottleneck is the database itself, as it is now the central point of failure and heavy load. This leads to the same problem concerning bottleneck and failover as described above. In other cases, the servers may **synchronize** their memory directly. Whenever a server has new information, it must communicate this new information to the others. When the server is required to delete data, this server needs to order the other servers to delete this data too (see fig 1.4)

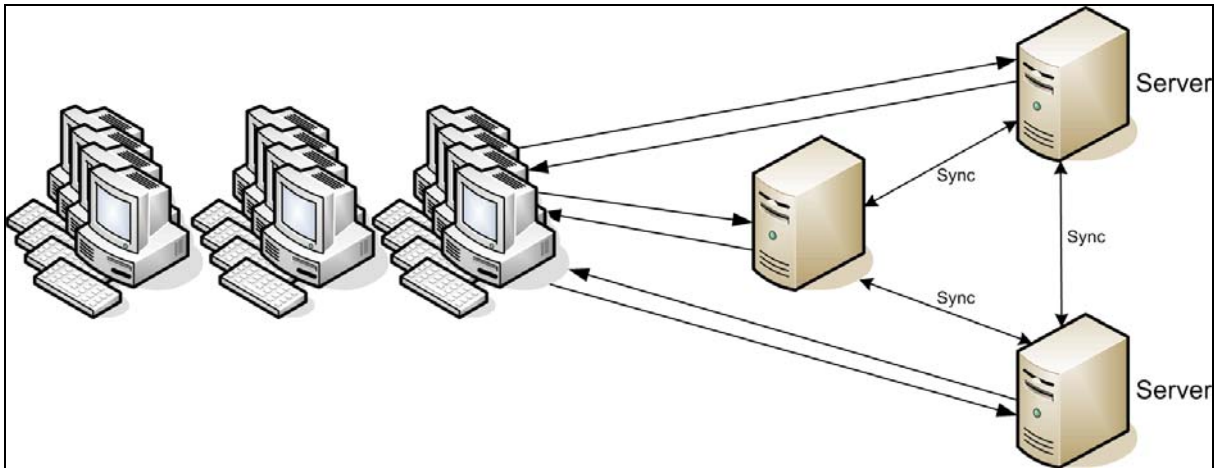


Figure 1.4: Synchronization of a set of servers

However, still the problem remains that servers may fail or become overloaded.

## 1.2 ... to p2p networks ...

Recent technologies [1] in offering information for programs are not using dedicated central servers, but instead, each participant in the network is a "peer". Therefore such networks are called peer-to-peer (p2p) networks. In such a network, each participant may act as a server and as a client at the same time, in decentralised p2p overlay structures these peers are also called „SERVENTS“ (SERver + cliENT). To be able to answer to requests, the peer must have at least a part of the complete information that might be needed by other peers. A peer can ask a set of peers and try to get the desired information. It depends on the replication and caching strategy of the p2p network [1], how to obtain the data and when to give up, when it seems that the information cannot be obtained within reasonable time. For example, Gnutella [2] asks a group of so-called Ultrapeers that don't necessarily contain the searched data themselves, but store information about successful search requests to peers. Therefore, not all peers need to be asked for information, but it is sufficient to only request Ultrapeers, which in turn ask the so-called leaves, which don't cache the request's responses. The Ultrapeers are iteratively asking other Ultrapeers for the requested information until a certain threshold.

### 1.3 ... to Space-based data storage

Obviously, it would be nice if **all peers** had the **same information**, so either information is held by a peer itself (don't forget that the peer acts as a server as well!) or the information is not available at all – i.e. no other peer needs to be contacted to ask for that information. In most cases, it is impossible to use such an approach, as with the just-mentioned file sharing network, as there would be too much data to be distributed between and stored at the peers.

But this scenario has some interesting advantages as it solves the problem with failing and overloaded servers: if one peer fails, availability is still given. If one or several peers are overloaded, only the user requesting the overloaded peer is affected.

Such functionality can be offered when using space-based data storage. All data is stored within a so-called data space, which the other peers can access. The data space may be spread among a set of peers, which need to keep the data in sync (which is called "replication"). The space only stores the data objects and manages the requests for such data. It also may store the data to a persistent storage device to recover from failure or to enable restart of the system without loss of data.

There are several implementations of such a data space existing, for example Corso [12] JavaSpaces [3], GigaSpaces XAP [4], TSpaces [5], XMLSpaces [7] and XVSM (eXtensible Virtual Shared Memory), although not all of these spaces provide replication or persistency of data:

<b>Name</b>	<b>Replication</b>	<b>Persistency</b>	<b>Transactions</b>
Corso	X	X	X
JavaSpaces		X	X
GigaSpaces XAP	X	X	X
TSpaces		X	X
XMLSpaces	X	X	X
XVSM	P	P	X

Table 1.1: Comparing several data space implementations. X means that this feature is offered, P means this feature is planned

One major advantage of XVSM is what the X in the name is standing for: it is standing for eXtensible, which means that XVSM can be extended by several

functions, like automatic data persistency and recovery, accessing the data by other selectors than the default ones, and so on.

In XVSM, the place where data entities are stored in is called "Container". One peer can host a set of Containers, this set is called local Space in XVSM:

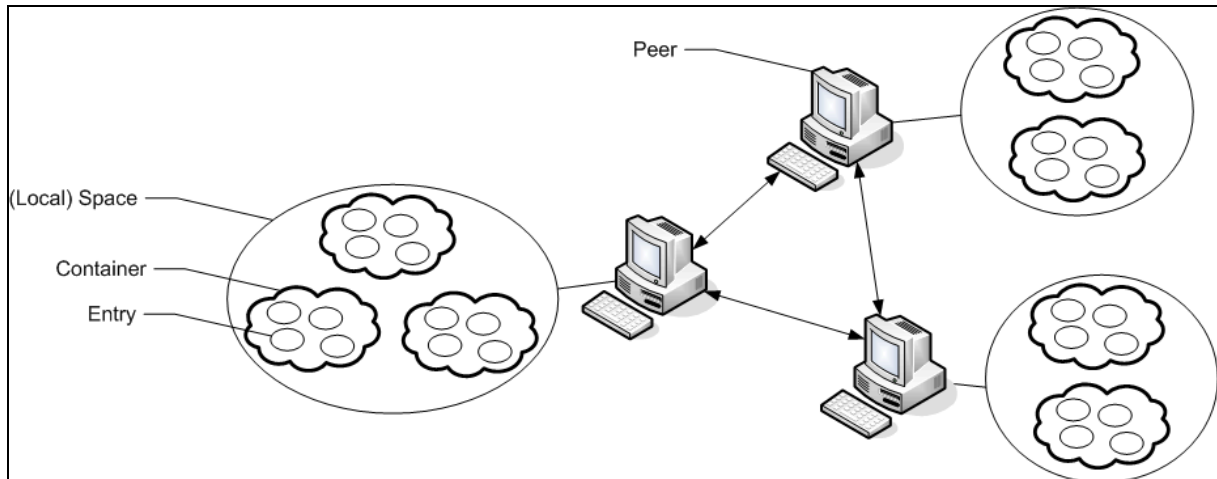


Figure 1.6: p2p Network, each with its own Space and its own Containers and Entries.

Data entities that are stored in a Container are called "Entries". An Entry can be either an AtomicEntry that is generic and can be instantiated using any Java class or a Tuple, which can contain a (set of) Tuple/s or AtomicEntry/ies. The Entries are written to, or taken, read or destroyed from the Container:

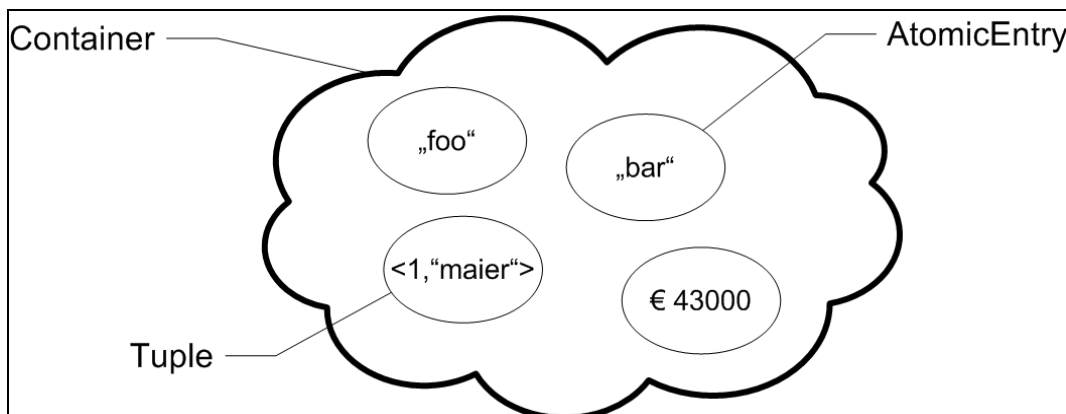


Figure 1.7: Overview Container, Tuple, AtomicEntry

As you can see in fig. 1.7, the Container is used as storage for data entities, which can be considered messages: one peer writes to the Container and wants to tell the other peers something by doing so. Thus, XVSM handles the communication and you no longer need to handle it "manually". The distribution of data is done by XVSM. Note that the data that is stored in the Container can be payload data as well as

command data, even mixed in one Container. Further note that using this approach, you can achieve decoupling in the sense of time and space.

As other peers might need to know about data stored by a peer, they are listening to new information. Such a listener on new information is called in [11] "Producer", which is listening to requests from a peer called "Consumer" [13]. The Consumer writes an Entry to a container, and the Producer takes it. As an answer, the Producer writes an Entry to the container that is taken by the Consumer:

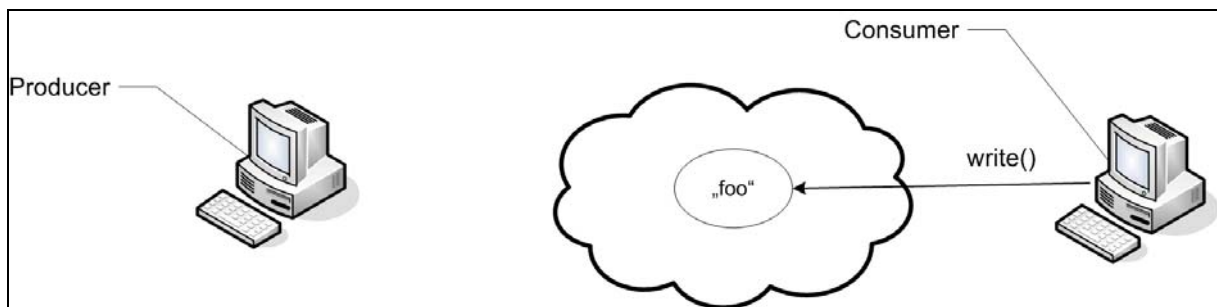


Figure 1.8a: Consumer writes data

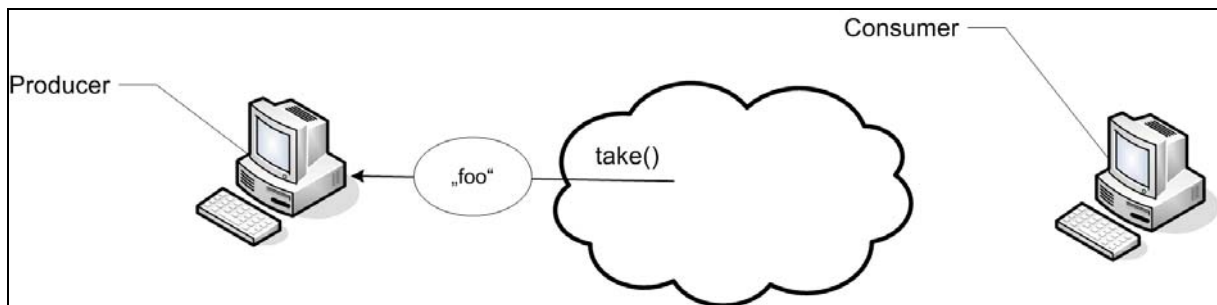


Figure 1.8b: Producer takes data

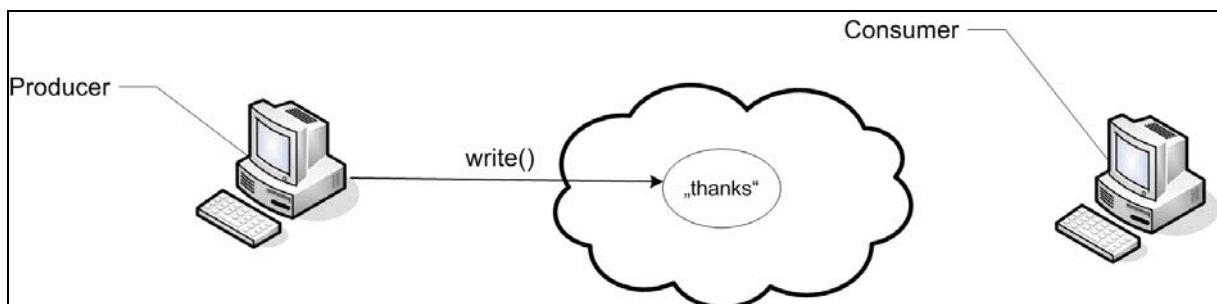


Figure 1.8c: Producer writes data back

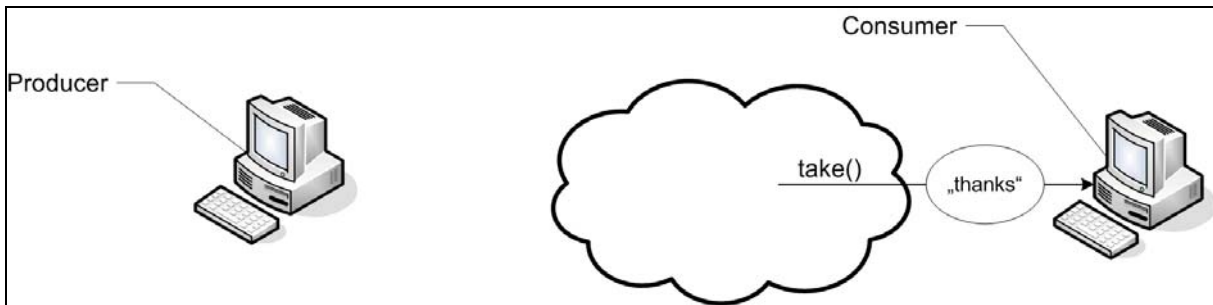


Figure 1.8d: Consumer takes data

XVSM offers the possibility to notify the peer that is working on the Container if a certain Entry is written to the container. Using this functionality, one can implement to push data by using notifications or to regularly pull data from the Container. It is no problem to also mix these approaches within one Container.

### 1.4 „Being more honest“

Ralf Westphal explains in [11], that techniques like RMI or CORBA hide the important fact that the producer and the consumer are physically distributed – a call to a remote process just looks the same as a call on a local object, i.e. in the same context:

```
String returnCode = producer.foo();
```

Internally, for the treatment of a remote execution, the consumer's method call is given to a communication proxy (e.g. RMI stub), which creates a communication package, and sends it to the producer's communication proxy. This proxy then hands the method call to the producer:

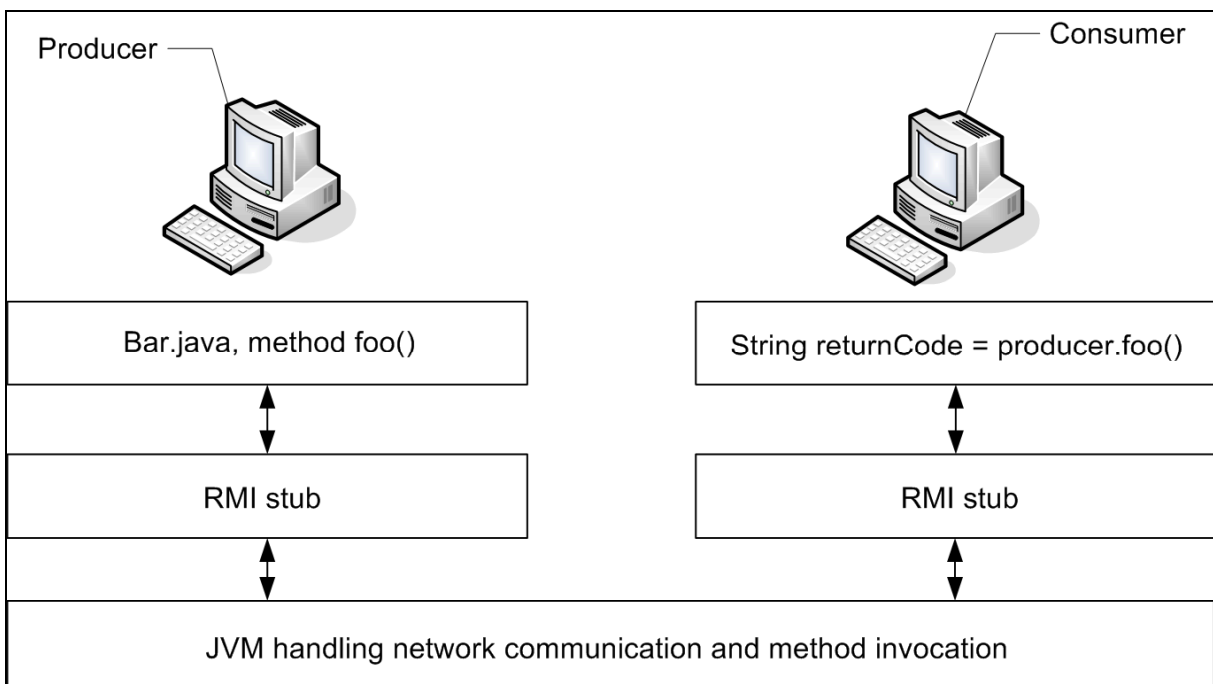


Figure 1.9: Simplified function of RMI

But hiding the details of the communication might lead to wrong expectations at the caller's side and thus according to [11] to "wrong promises". The reasons are:

If you want to perform a local method call, you can be sure of the following assumptions [11]:

- Execution of the request and delivery of the result without any delay
- Handing over complete control of the thread to the producer
- 100% availability of the producer
- 100% secure communication between consumer and producer
- 100% fail-safe communication between consumer and producer
- The execution is performed immediately

Obviously, these assumptions are not necessarily correct for working with remote processes. But as remotely running processes should act autonomously (according to [11]), there should be another paradigm used, and that's where XVSM comes into play: XVSM can be used as a message exchange platform, which holds the requests from the consumers and the answers of the producers. The consumer cannot "command" the producer directly, as the consumer only writes a request to the space and the producer takes the request to fulfil it. The producer autonomously decides to handle a request and it also decides the order of handling requests; this style of interaction is not *imperative* anymore, but *cooperative*:

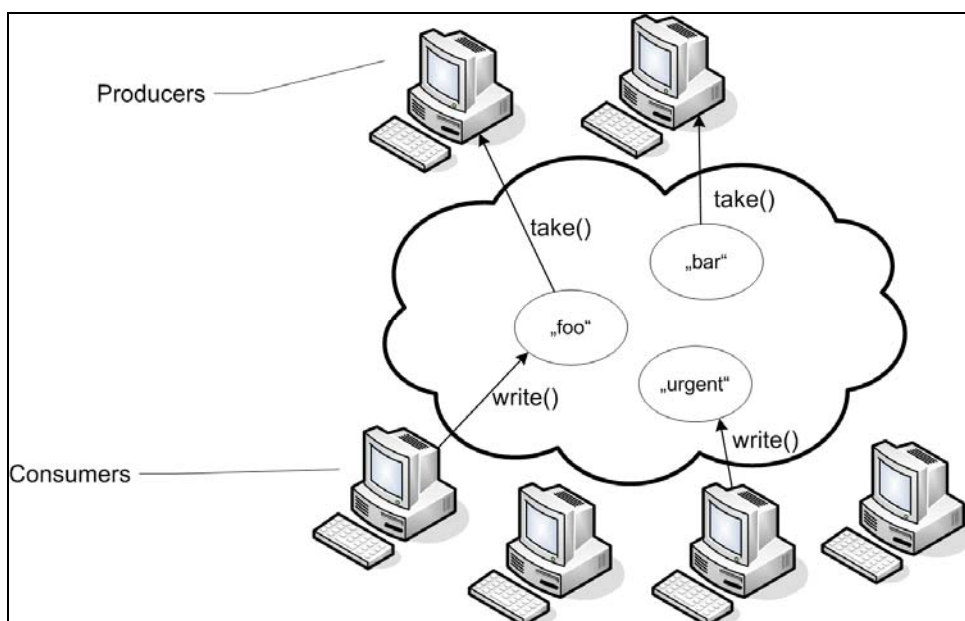


Figure 1.10: Multiple consumers, multiple producers

As XVSM, like other space-based approaches, offers the possibility to let multiple peers communicate using the space, there might be multiple producers handling the requests, and they can cooperate to handle them. On the other hand, the detail that a request might be lost, as no producer is available, is not hidden from the Consumer – it has to provide some possibility to be notified as soon as the request is handled. This might take forever if no producer is available for this request, but it is on behalf of the consumer to take special treatment in such a case, which is more appropriate for operating remotely and is not hidden from the programmer.

## Chapter 2: Usage Possibilities for XVSM

### 2.1 Adding more computation power

In chapter 1.4, the possibility to use XVSM to implement an improved version of the Client-Server architecture towards a “more transparent” way of request processing was shown: the Client is aware of the possibility that no server is available and thus it may take action that its request will be fulfilled within a given time. In fig. 1.10, of course there is still the possibility that Producers fail or that they are overloaded. But as Customers are aware of this behaviour, they can either try again or contact an authority to ask it to add more Producers:

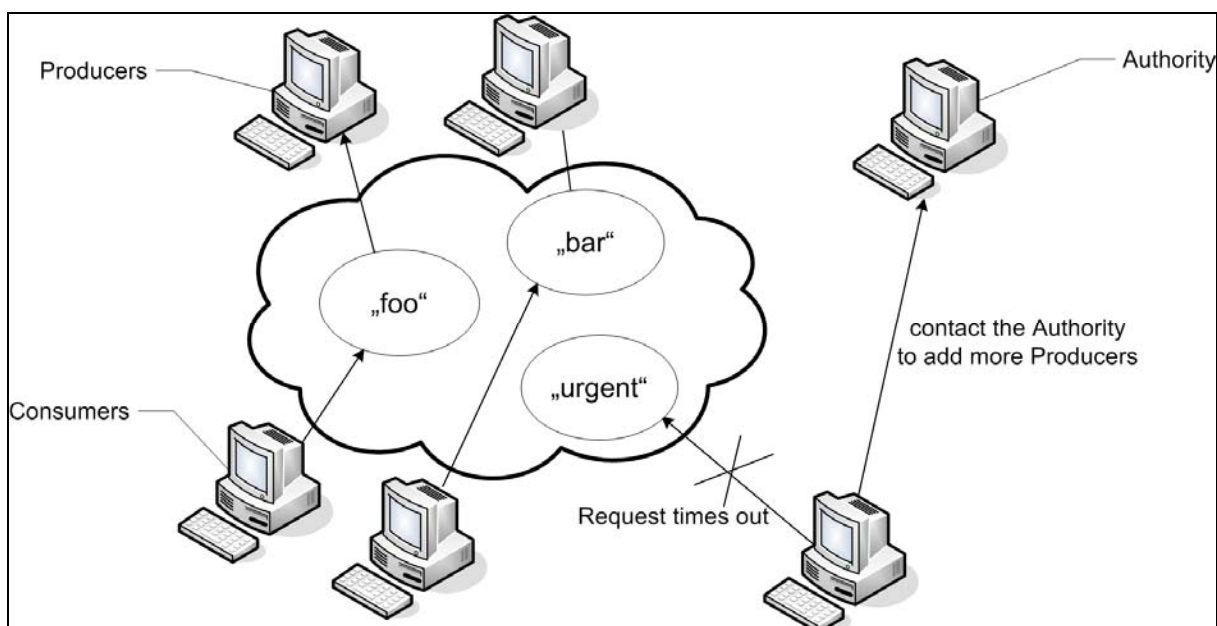


Figure 2.1.1: Request times out, Consumer contacts Authority

The contact to the Authority can be established according to certain policies like e.g. after a certain number of retries. The Authority decides if it is really necessary to add new Producers; e.g. if all Producers are really overloaded. In the case that the Producers are not fully utilized, but the Consumers complain that the Producers are not answering, it might be that the Consumer(s) are not requesting in a way that the Producers are able to react upon. For comparison: Imagine a room with two sales persons and several clients. The sales persons answer questions about product prices. However, assume that there is one client that asks when the bus will arrive. The sales persons won't answer to this question and after a few retries, this client will give up. An Authority now may decide that it's not a problem of the salespersons (as they are not overloaded), but that the client is not requesting correctly only by comparing the utilization of the Producers (which indicates that they are not overloaded) and the fact that requests from Consumers are arriving that there are not enough Producers. In this case, the Authority must provide some means to contact a human administrator to inform about the fact that a Consumer's request cannot be fulfilled. This is in contrast to an imperative approach: Using an imperative approach (e.g. Client-Server architecture), you receive an answer either way, which can be an error message if the request was formatted in a bad way. In some cases, also the cooperative approach (as just described) can handle cases where erroneous requests are detected, but only with limitations. If a request is too incorrect, it may not be processed at all.

In case that the Authority decides that a request is invalid, it can remove this request from the "normal" container and move it to an extra "error container", as proposed in [15]. The human administrator may be notified about such a new Entry or he/she may check this containers content regularly to decide what to do with such Entries.

It depends on the implementation of this Authority, when it decides to add new Producers, and how it is done – but let's say there is a pool of available peers ("Resource pool") and one peer is added to this space just by telling it that it needs to handle (a subset) of requests and giving it the reference to the space container where the requests are stored. The Resource pool may be the internal memory of the Authority or another container with contact information to available peers. When writing this information to the memory or to the container, the peer is considered being available for a new execution task.

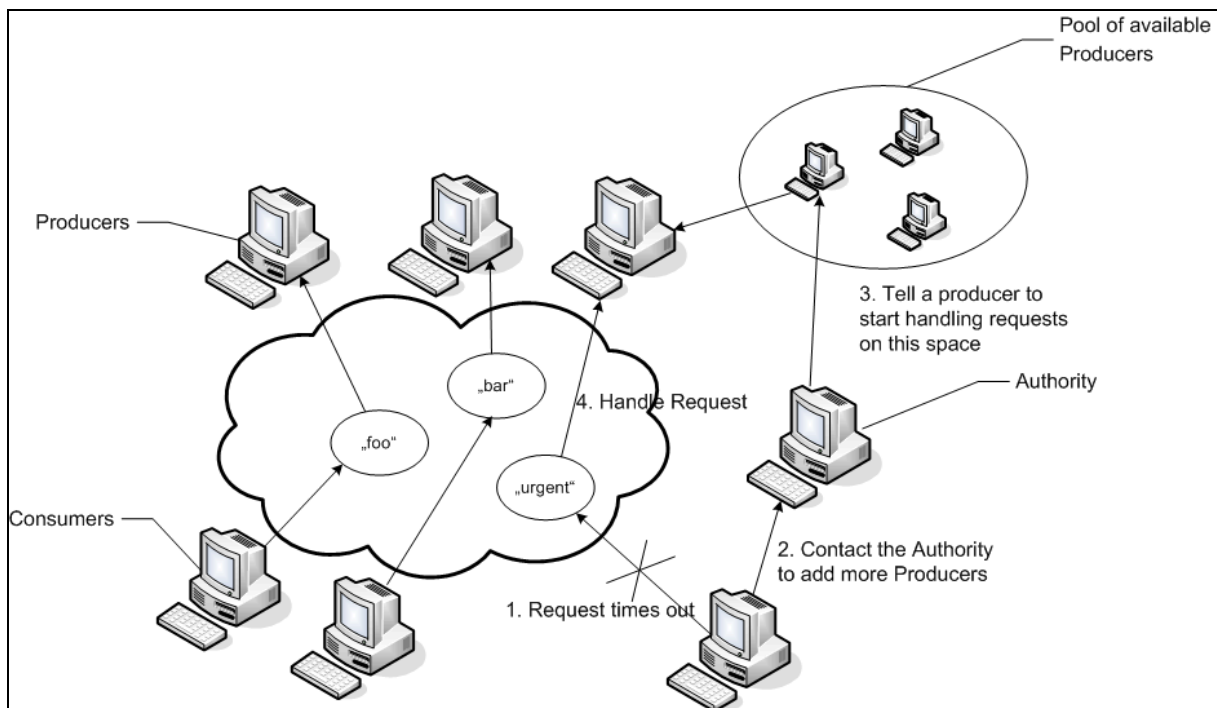


Figure 2.1.2: Adding a new producer

A very interesting property of this scenario is shown in fig 2.1.2: A Producer actually can be a thread in a process that is running as a Producer, a single workstation, a mainframe or a cluster that is running processes that are acting as Producer – it just doesn't matter. It is always only a “worker” that is checking if a certain request is standing in the Container. Of course, if there is no Authority or if there are no more producers available, also XVSM can't help. But keeping in mind that XVSM allows to detect whether a request is not handled, the customer can – in contrast to other paradigms – handle such a situation. For example, when an Entry is currently handled, it is taken from the Container and written to another. If after a certain time, the Entry still is in the Container (it is still not processed), the Client that stored the Entry can take action, like error handling, user notification etc.

## 2.2 High-priority requests

An Entry may hold data beyond or instead of a request itself (called payload data): so you can use this data to define a priority. It is also possible that the Entry itself is of a type (class) that represents a request with high priority, for example a type that extends the functionality of the Entry by giving it a "high-priority label". With such a

request that is declared as being high-priority, you may implement a system where a dedicated peer listens only on requests with high priority:

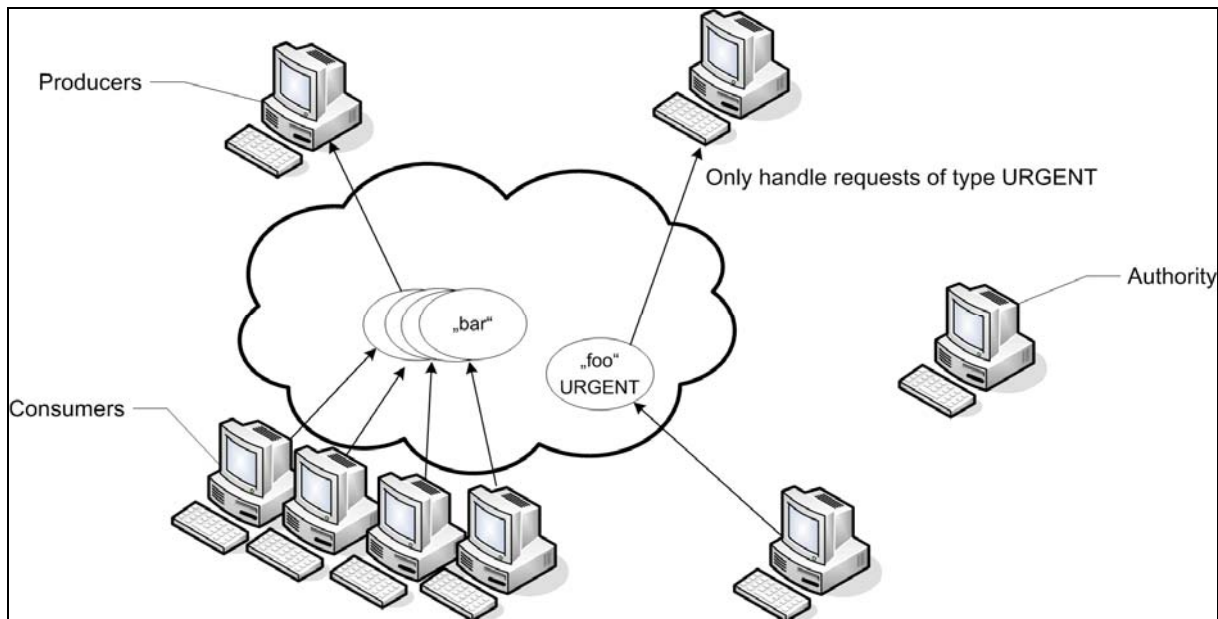


Figure 2.2: One Producer only handles urgent requests

There is also the possibility to implement the high-priority request handling not with a dedicated producer, but in a way that each peer (before taking an Entry from the container) checks if there is a request with high priority. If there is one, the request is executed, if not, a "normal" request is taken.

## 2.3 Execution path

Building on the examples from Chapter 2.1, we can implement an execution path. An execution path is a sequence of operations, where the output of one operation is the input of the subsequent operation. It has one or many defined Start state(s) and one or many defined End state(s).

We could define an execution path as the following state machine:

- There are Pools that contain intermediate results of an overall operation. The Pools are represented by a state in the diagram.
- The edges are operations that are performed by one or more executing entities on an Entry in the Pool. Such an executing entity is a peer in the space that offers its processing power to perform the operation.

- An operation moves an Entry, which contains the intermediate result of the operation, from one state to the next.
- There are Pools at the start and at the end of the path that contain the initial values resp. the final results.
- The sum of all passed edges between a start and an end Pool are considered as being a complete computation (e.g. the computation of PI, or fulfilling a Sales request etc.).

An example for an execution path could be:

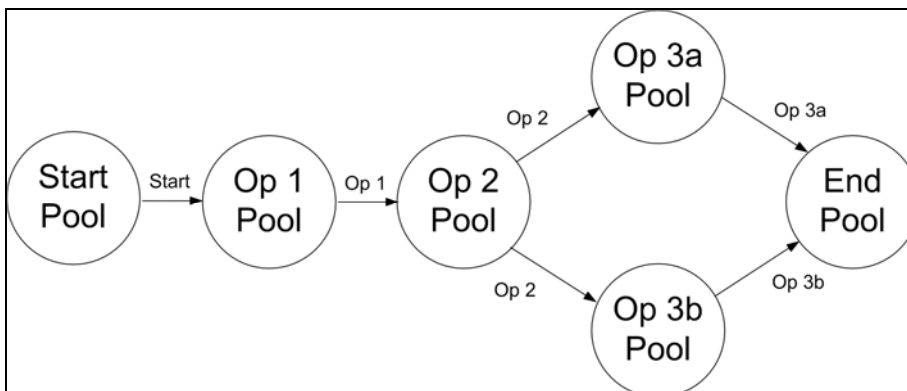


Figure 2.3.1: Execution path

In fig. 2.3.1, the Pools are drawn as circles (states). In the Pools, the above mentioned Entries with the intermediate result are stored. The linking arrows between the Pools are standing for the operation that is needed to bring an Entry from one Pool to the next.

On each operation, some entities are included that perform calculations (processors, clusters, ...). After an operation is finished, the result is passed to the next Operation Pool. There also may be – depending on the result of an operation – a branch in the Execution path. Furthermore, there may be multiple start and/or end states:

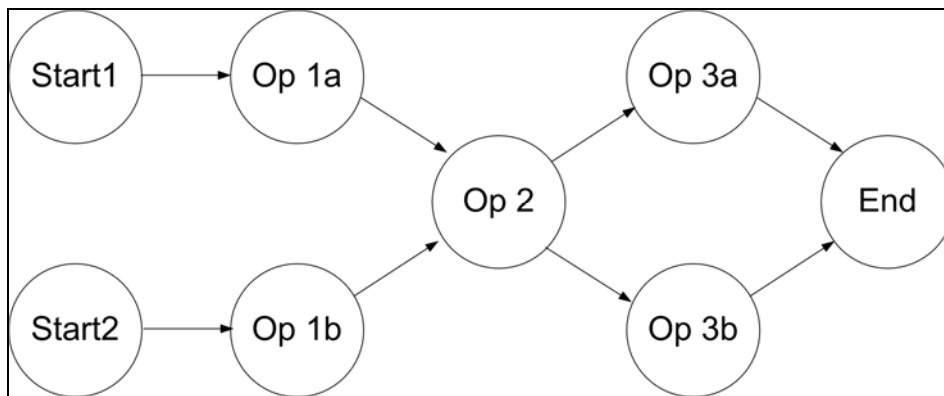


Figure 2.3.2: Execution path with multiple Start states and branches in the calculation from Start to the End

The possibilities to combine the states (operations) are vast.

If you have a calculation to perform, you put your initial value to (one of) the Start states. A calculation entity takes this value and performs an operation on it. The result is passed to the next Operation Pool. You can imagine this taking and passing of the value like a token that moves from one state to the other in the above state diagram, which is also known as the SEDA approach [6]. One of the advantages of the SEDA is that multiple peers can work on one or different states in parallel and independent from each other. Thus, in heavy load conditions, it can perform better compared to a thread-based approach, where each thread executes the complete execution path without any intermediary states [6].

Let's say that the token that represents an intermediary result is an Entry in XVSM and each of these pools which hold the intermediary results is a container. Peers are listening to new Entries in a specific container. A peer takes an Entry and performs an operation on it. When it has finished, the peer writes the result to the next container (state). There, it is taken by a (possibly other) peer and so on, until all operations are done and the result value is written to the End state.

Each container holds a number of Entries, which are the intermediary results. One can now check these numbers per container to see the current progress of the calculation and also see the number of peers that are working on this container. A tool that visualizes the number of working peers and number of Entries per state is implemented in [14].

Obviously, each operation needs at least one peer to perform this task, else, there would be an increasing number of Entries in the container without a calculating peer

and there won't be any result. But of course there could be multiple peers operating on one container:

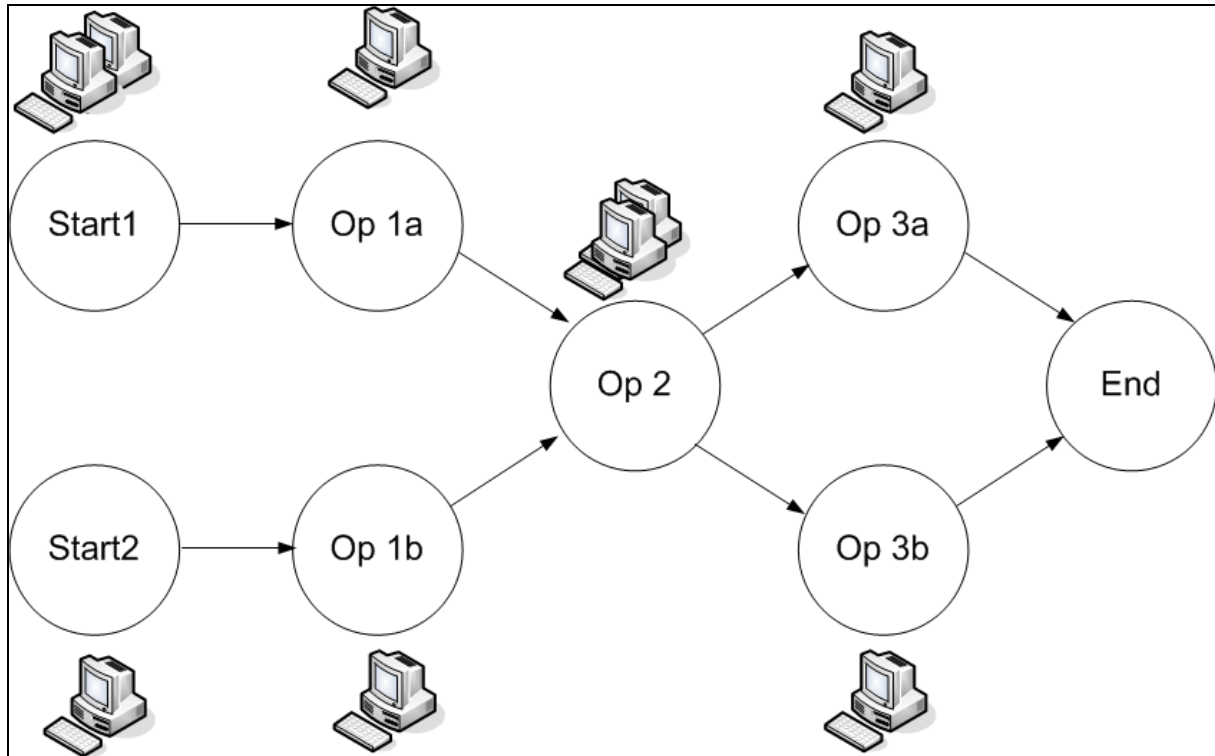


Figure 2.3.3: Execution path with multiple peers working on one container

The Entry already contains the algorithm as additional payload, for example as a script. Thus, the peer takes the Entry and performs the operation that is common for this one container. The information which operation is specific for which container may stand in the contained algorithm. In this simplified example, it is only assumed that there is some possibility to store the algorithm within the Entry.

Keep in mind that an Entry could be an AtomicEntry, which is generic. In this AtomicEntry, you have some object stored, that holds the current value and describes the operations as well as what to do if an operation is finished, i.e. which operation should be done next. Beside the complete algorithm it furthermore has an internal state that tells which operation should be currently performed. Peers participating in this Execution path only need to retrieve the algorithm that is specific for this container and check what to do with the Entry when the operation is finished, which must also be included in the Entry's algorithm.

Please take a look at Figure 2.3.2. A peer takes such an Entry from the Start 1 container and executes the algorithm contained in the Entry. The algorithm must describe which operation to execute first. This algorithm is executed by the peer. When the algorithm that is specific to the current state is finished, the internal state of the Entry is set to let the second method perform next. But this method is not performed by the peer immediately. Instead, the peer checks by usage of the return parameter of the current algorithm that the Entry must be written to the subsequent container. Thus the peer does what the Entry tells it and writes the updated Entry to the Op 1a container. There, the Entry is taken by a peer (either the same or another one), the algorithm which is specific to container Op 1a is called and so on.

Each container now holds a number of semi-products (Entries where the calculation is not finished yet). As the peer doesn't know the algorithm in advance and doesn't know what kind of Entries are standing in the containers, the peer doesn't need to be prepared specially. It just needs to execute the algorithm passed with the Entry. All the operation instructions are stored in the Entry itself, the peer just performs the operation. Using this approach, you can use a peer for any operation state, as the peer doesn't care what container or Entry it is currently working on, it only performs the actions told by the Entry. Security issues are not taken into consideration in this simplified example, but of course may be an issue in a real-life project.

## ***2.4 Execution path with load balancing***

Let's assume we assign exactly one peer to each container. Then, as already explained, the Entry is passed from one peer to the next. Furthermore, we assume that each operation takes equally long and that there is the same number of initial Entries written to Start 1a and Start 1b. But in Op 2, there is only 1 peer listening for new Entries. Thus, if the Entries are coming in quite fast, this peer will be overloaded and it cannot perform the operations in time. The preceding peers are writing their results to the Op 2 container and thus, this container gets more and more Entries.

Please keep this scenario in mind as it will be referred to again in Chapter 2.5.

We already know from Chapter 2.1 that with the usage of a Resource pool and an Authority, a new peer can be added to "support" other peers with their calculations. The Authority regularly checks the number of Entries in each container. If there is a high number of Entries in a container that exceeds a configurable threshold and if there are resources available in the Resource pool, resources are added in the form

of peers to the container. Note that such a processing resource could be of different types (thread, processor, mainframe etc.). The processing resource (a.k.a. peer) is told that it now is listening to Entries in a specific container. As soon as an Entry is written to the container, the Entry must be taken and its algorithm executed. In the scenario just described, where in Op 2 an increasing number of Entries will occur, it is appropriate to add a peer to this container to perform the operation. It is up to the Authority to decide, (a) which peer should be added to the container, and (b) at what load per container the peer should be added. Of course, if there is a low number of Entries in a container, the Authority could decide to take peers away from the container and give them back to the Resource pool (“shrink” vs. “expand”):

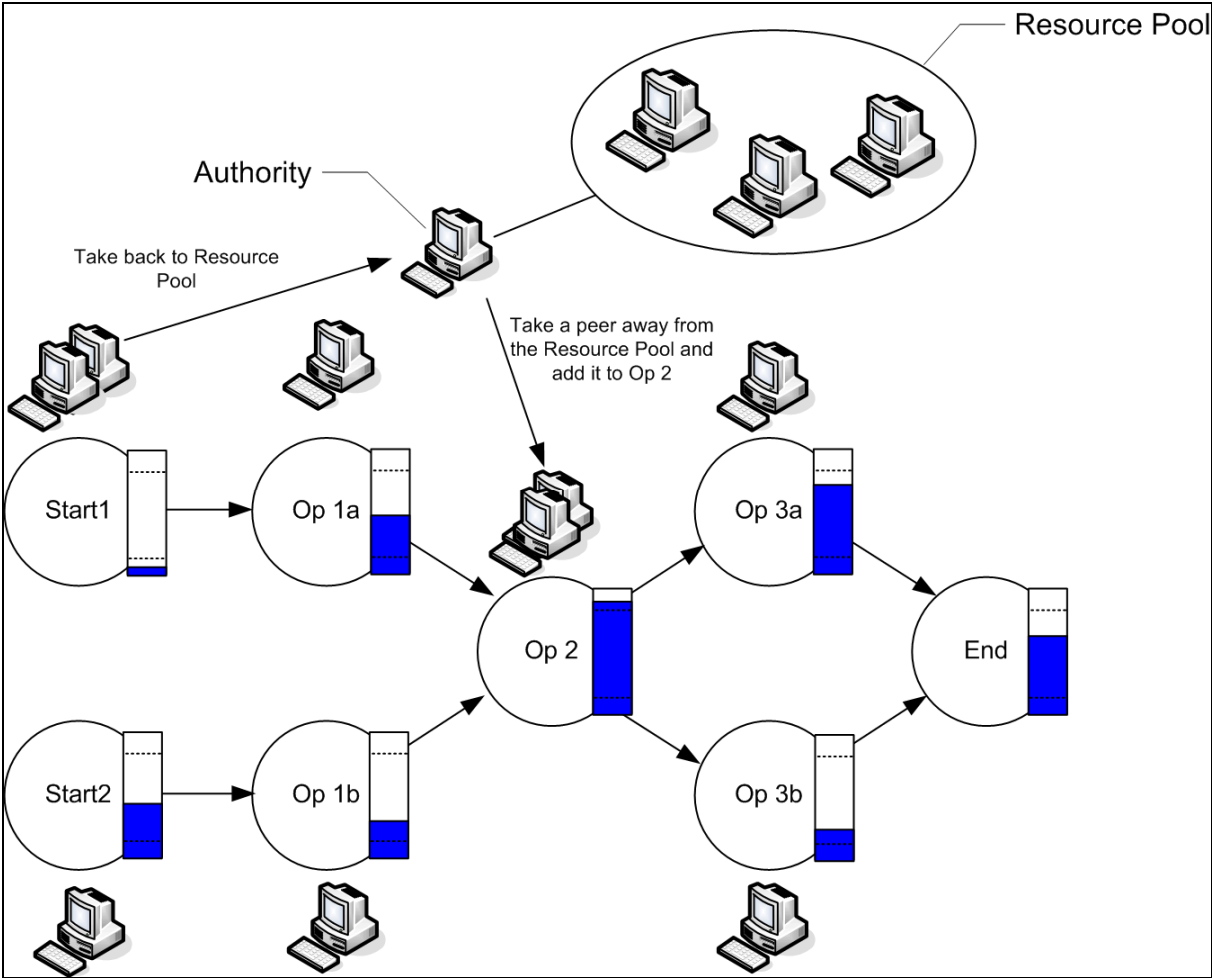


Figure 2.4.1: Execution path with an Authority to perform load balancing

The symbols used in fig. 2.4.1 and subsequent figures represent the following:

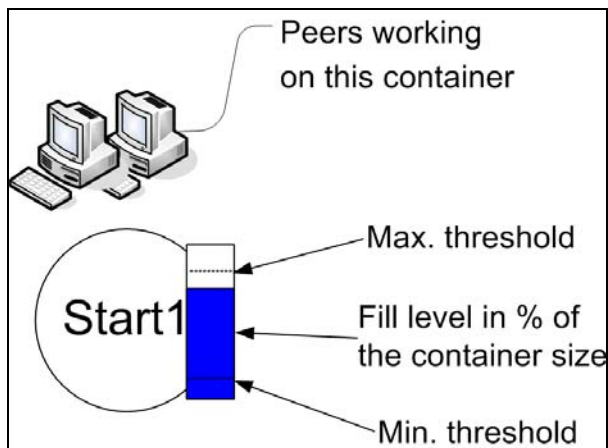


Figure 2.5.1b Symbols used in the previous diagram

Again, please note that the resources, the container and the Authority are independent of the operations that are done and of the Calculation itself, as all the information about the operations and calculation is contained in each Entry. This scenario can of course also be accomplished by many other programming frameworks. But XVSM has an advantage in comparison to the some other ones: an Aspect can be implemented that performs the necessary calls to listen to a container, executes the Entries' algorithm and writes the Entry to the container of which the Entry knows that it needs to be written to next. You register this Aspect at an XVSM peer that wants to join the Execution path and that's all. Other middleware systems that offer similar functionality to implicitly react on performed actions is for example Lime [19], which offers the possibility to register reactions on actions that are performed on the space.

The Authority can be implemented in a way that it is called by a peer, if the peer notices in some way that the upper/lower threshold of container entries is exceeded that shall trigger addition/removal of Entries in a container; or the Authority can check the various containers time by time by itself and decide if peers should be added or removed from a container.

We assumed that each operation takes equally long and that into Start 1 and 2 the same number of initial Entries is written. But it is quite obvious that this may not be realistic. operations can have different duration and the initial values are not written with same frequency. But this doesn't matter: Let's say that there are twice as much Entries written to Start 1 than to Start 2. On both containers, only one peer is

listening. After some time, there might be so many Entries in Start 1, that the Authority adds one more peer to Start 1. Then, there will soon be too many Entries in Op 1a, so the Authority adds here one more peer too, and so on. After some time, let's assume that no more Entries are written to Start 1. There are currently two peers working on this container, so eventually there will be no Entries in this container any more. These two peers notify the Authority that the container is empty, so the Authority withdraws one (or even both) from the container and returns them in the Resource pool as described in Chapter 2.1. When the peers are in the Resource pool, they in turn can be added as a worker to another container again.

If you want to implement the execution path, it is not necessary to have a single container for each intermediate result, but you may have only one container for all Entries, no matter what internal state they have. Peers accessing this container are filtering the Entries by their state and taking those Entries that they are listening on. For simplicity reasons, the style with one container per intermediate state, as already described above, is used.

## ***2.5 Execution path with automatic load balancing***

One thing that is still needed in the previous scenario is the Authority that adds or removes the peers from the containers. But it is possible that the peers are managing this by themselves.

Take the same scenario as from Chapter 2.4, but without the Authority, and assuming that the containers are limited in size. There is only one well-known peer running that holds a container that represents the Resource pool. If a peer decides to request more computing power for the container it is currently working on, it writes a request to the Resource pool (which is a container, too). If a peer decides that there are enough working peers on its container, it could decide to "leave" this container. In this case, it gets one Entry from the Resource pool, which is a Request for Help. From now on, the peer listens to Entries in the container where help was requested:

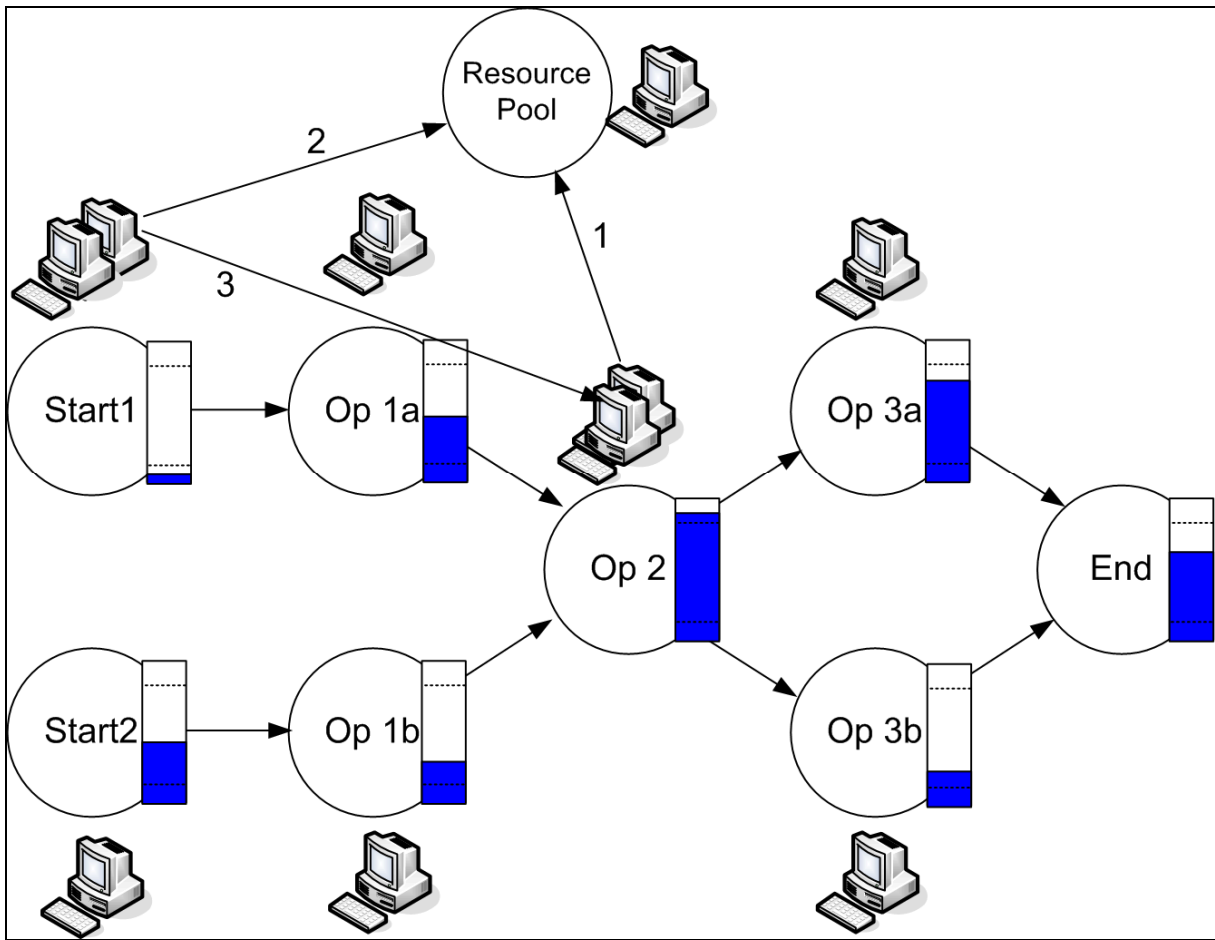


Figure 2.5.1: Execution path with different fill levels of the containers

1. The maximum threshold is reached, so the peer that is working on Op 2 writes to the Resource Pool an Entry that holds a request for help. The request for help must be unique for a container (usage of keys), else many peers per container could write the same request many times to the Resource Pool.
2. A peer decides that the minimum threshold is reached and thus, it doesn't need to work on this container any more. So it reads the requests in the Resource Pool. If there isn't any request, the peer either blocks (as it is done if there is no Entry in a container) until a request is written or it returns to its old container, waits for Entries and regularly checks if there are new requests in the Resource Pool. But as in our example scenario there is a request in the Resource Pool, the request that Op 2 needs help, is taken.
3. Thus, the peer joins Op 2, and starts taking Entries from there and executing them.

The approach described so far has two – yet – unsolved problems:

- The peer that possesses the Resource Pool crashes
- "Elopement from an empty container": All peers on a container notice more or less at the same time that the minimum threshold is reached and so, each of them retrieves a request from the Resource Pool, so no peer is any longer working on this container.

The first problem can be circumvented by detecting the crash of the Resource Pool peer. This is not hard when using a separate container for the Resource Pool, as a peer that wants to contact the Resource Pool will notice that it is unavailable by the standard network exceptions. In this case, the peer that detects the failure of the Resource Pool will take over its role by creating the Resource Pool container, which must be well-known by all participating peers. If you decide not to use separate containers for each Operation Pool but instead to write all Entries to one container, you also may decide to write the Request for Help Entries to this container, too. In this case, there is no network exception because the container representing the Resource Pool is part of another container:

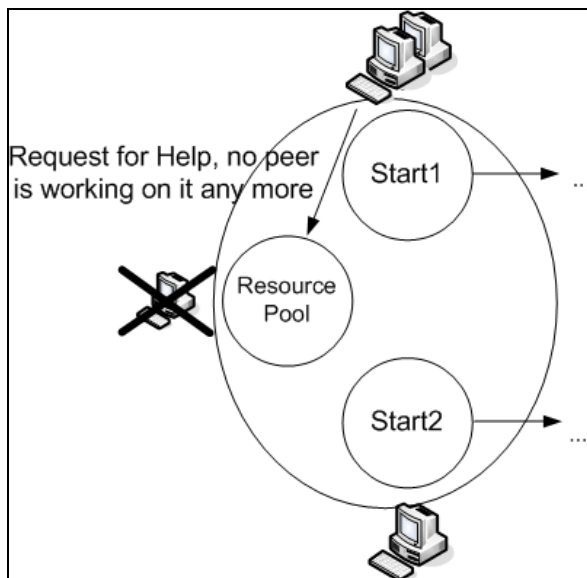


Figure 2.5.2 Peer that handles Requests for Help has crashed, the container itself is up and running

In this case, there is no dedicated peer that only works to handle the Requests for Help messages which could crash. The peers themselves are writing and taking the Requests for Help, thus the peer holding the Resource Pool only needs to offer a container where the Request for Help messages can be stored in. To conclude, if the

container containing the Request for Help messages crashes, then there is a problem, but this can be circumvented as it is easy to detect, as just described.

The second problem, where peers may elope from a container that has reached a minimum threshold can be fixed in the following way: Instead of deciding that the peer itself leaves the container, it stores in this container a message to tell the others that they should leave. Every peer that is noticing the minimum threshold is reached needs to check if such a message exists, except the one that just wrote this message to avoid to let the peer that wrote the message also read its own message. There must only be exactly one or none of such an Entry (by using keys for example), else it might be that all peers at once write such Entries and then none will leave. So, before writing an Entry to leave the container, a peer needs to check if such an Entry already exists and if so, it needs to take this Entry and leave the container.

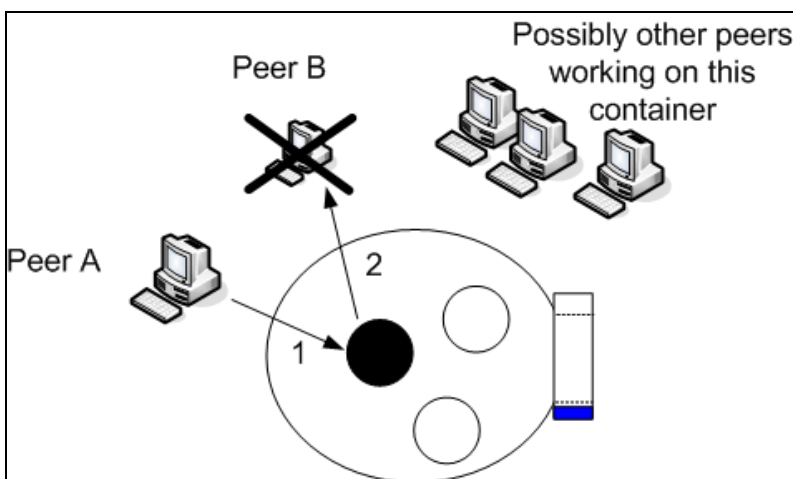


Figure 2.5.3 A peer notices that the minimum threshold is reached and writes the "Leave container" message

The black circle is the unique message (Entry) to "Leave container". The white circles are the Entries to perform the "normal" operation.

1. Peer A notices that the minimum threshold is reached. It checks if the "Leave container" message exists, but as it doesn't yet exist, it writes this message. It no longer listens on a message to leave; else it might be that it receives its own message.
2. Peer B also notices that the minimum threshold is reached. It checks if the "Leave container" message exists. As it arrives (created by Peer A), Peer B

- destroys the message and leaves the container (it is returned to the Resource Pool or retrieves a Request for Help, dependent on the approach you chose)
3. The other peers are not included in this scenario, but they might perform similar actions as described in point 1 and 2.

Still, the following scenario could occur:

1. Peer A and B notice that the minimum threshold is reached.
2. Peer A obtains a lock on the container and writes the "Leave container" message.
3. Peer B also wants to obtain this lock, but it is blocked until Peer A commits and unlocks
4. Peer A is no longer listening on "Leave container" messages, else it may be that it receives its own message, which may lead to Elopement again.
5. Peer B notices that there already is a "Leave container" message, takes it and leaves the container
6. Peer C notices that the minimum threshold is reached, sees that there is no "Leave container" message and thus writes one, and furthermore is no longer listening to these "Leave container" messages.

In this scenario, Peer A and Peer C are still working on the container but no longer listening on the "Leave container" messages. To prevent this behaviour, the Peer that no longer listens to a container checks in random intervals if there is such a message. If so, it destroys the Entry and listens again on it. If no such Entry is available, it just listens on its occurrence again to give another peer the possibility to tell to leave this container. By regularly checking the minimum threshold and performing the described tasks, the number of peers working on this container will not immediately decrease to 1, because of race conditions on the regular checks and on taking the entry, but in this case this is an advantage, as then a set of peer remains working on this container until an optimal balance is reached.

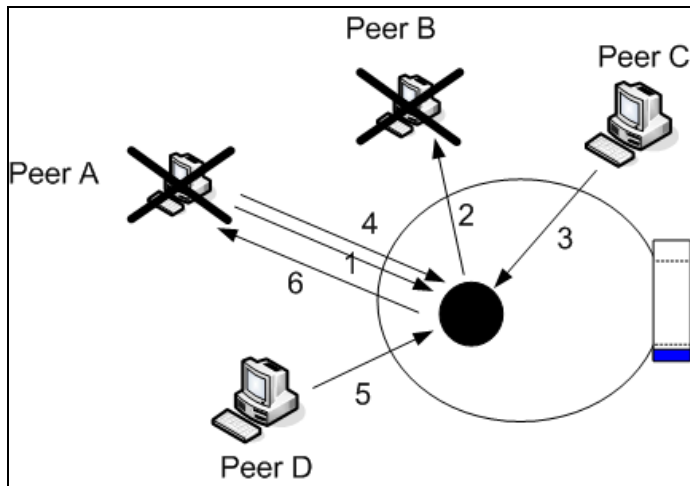


Figure 2.5.4 Peer A first writes the "Leave container" message but later listens to it again

1. Peer A writes the "Leave container" message, doesn't listen on its appearance any longer.
2. Peer B notices the minimum threshold, checks for the "Leave container" message, destroys it and leaves the container.
3. Peer C also notices the minimum threshold, checks for the "Leave container" message (which already was destroyed by Peer B) and thus it writes a new "Leave container" message. It also is no longer listening for this kind of message.
4. Peer A and C now check after a random interval if a "Leave container" message exists. Peer A checks first, it notices that there already exists such a message, so it destroys it and listens on these messages again.
5. Peer D notices that the minimum threshold is reached. It checks for a "Leave container" message which currently doesn't exist, so it writes it to the container.
6. Peer A checks for the "Leave container" message again, which now exists and thus it destroys this message and leaves the container.

As seen in the scenarios just described, a peer regularly checks in random intervals if the minimum threshold is reached. If so, the peer checks if there already exists a "Leave container" message. If there is already such a message, the peer destroys it and leaves the container. If there is no such message existing yet, the peer creates it. The peer sets an internal state that it already has written this message. After a random time, again, this peer checks if the minimum threshold has been reached. As

still, this is true, but the peer already has written the "Leave container" message (internal state!), the peer only checks if the "Leave container" message exists. If it does exist, the peer just destroys it. No matter if the message exists or not, the peer has no longer the internal "I've written a 'Leave container' message" state and thus is agreeing again to leave the container if such a message appears.

You also can implement the load balancing in a way, where the Requests for Help have a priority, depending at which level the maximum threshold was reached. Doing so, the container (state) that needs help most urgently is served first when assigning peers to containers.

## ***2.6 Execution path with recovery after failure***

Until now, we always assumed that each peer that is working on a container always works "perfectly" in means by its reliability and availability. But as we all know, programs and computers are not always working "perfectly". It could be that a peer is shut down during its operation or that someone writes malicious code in the Entry's contained algorithm that lets a peer crash. Also, network errors can occur that let a peer temporarily be unavailable.

Some problems can be solved using the scenarios that are described previously, others can't. In the following you get detailed information about what problems can be solved and how they can be solved.

The containers that represent the Operation Pools are limited in size. Thus, if a peer has finished its operation and wants to write the result to the target container, which is already full, the operation blocks. This means that also this preceding container becomes filled-up after some time, as the preceding peers are still writing to this container, but the writing peer is blocked. Eventually, from the container that was full, an Entry is taken. Thus, the blocked peer unblocks and writes its result to the container. It takes an Entry from its container, and the peers that were blocked because this container was full, are also unblocked and so on. So, if there is a blockage in a container, the blockage is accumulated to the preceding containers (see also "backpressure" in [6]). As an alternative, instead of letting the peer block,

we could let it detect the blocking via an exception (instead of using an infinite timeout) and let it write a Request for Help for the target container:

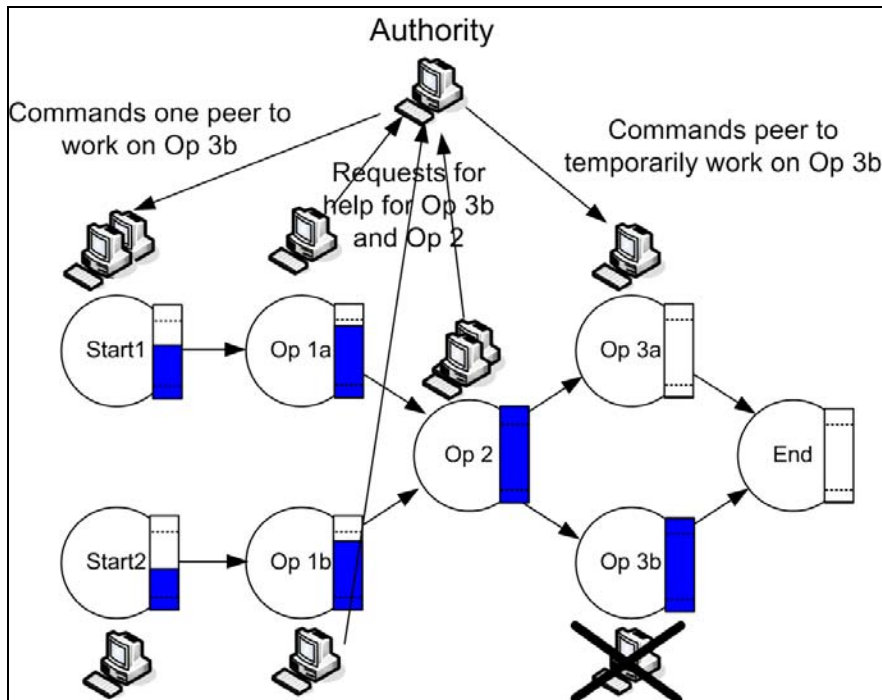


Figure 2.6.1 Accumulated blockage with Requests for help for the target container

As you can see, instead of the automatic load-balancing strategy explained in chapter 2.5, the Authority is used again, as it needs to decide what to do if the Resource Pool is empty. In this case, it needs to tell a specific peer to leave its container and work on the container that is full to resolve the blockage (“re-scheduling”). Remember that the peer that wants to **write** to the container which is full must contact the Authority, in contrast to the previous scenarios, where the peers that are noticing that the container they are currently **working** on, are contacting the Authority themselves.

This approach can be used in combination with the previously described automatic load balancing strategy: If the maximum threshold is reached on the own container, the peer writes a Request for Help to the Resource Pool. If a peer notes that the subsequent container is full, it writes a Request for Help for the subsequent container – remember that for the case the subsequent container's peer already has requested help, the request must remain unique in the Resource Pool, as already explained above.

If a single peer fails whereas there are others that are working on a container, it doesn't matter at all: The load of this container will most probably reach after some time the maximum threshold, a Request for Help is written and they will receive help, if available.

There are two problems unsolved:

If a peer fails while it already has taken an Entry from the container, the Entry is most probably lost. But the client that uses the execution path may be aware that probably none is working on its request and may take special actions to react on such situations. This is one of the advantages when working cooperatively instead of imperatively, that the Client is aware that the request wasn't successfully performed because the state is visible in the container. See also Chapter 1.4, about "Being more honest".

The Entry of the failed peer is not necessarily lost forever. Remind that the peer that is working on the current container might have a persistent container itself. Or it implements persistency by any other means, for example by writing the taken Entry to the file system. So whenever a peer takes an Entry from the container it is working on, it persists this Entry by some means. When the peer is restarted, before taking an Entry from the container, it checks whether there are Entries in its persistent storage and if so, these Entries are processed first.

If a peer that holds a container is stopped, all Entries of that container are lost. But if the peer that holds the container recovers, and if the container was a persistent one, all the Entries can be recovered.

Although these issues require some programming effort, compare this to the solution you have to design in the client-server architecture: you send the request to the server, if it fails, also here, your request and all its semi-results are lost. But using XVSM, you can at least recover the complete execution path after a failure of a single peer. In addition, patterns can be developed that support these recurring scenarios.

## Conclusion

There are a lot of possibilities that XVSM can be used for. Of course message queues and topic queues are alternatives to XVSM, which you can use to implement any desired functionality, but XVSM has the advantage that it offers a high abstraction through arbitrary access to shared data containers that goes beyond directed first-in-first-out access, and that it is extensible, so simply through adding a new Aspect, you can let your peer use load balancing or you can let it work with high-priority requests. By using the cooperative way of telling another peer to perform an action, the peer could either be a single processor core, a mainframe, a cluster, it doesn't make any difference, as there is no distinction made, they are just referred as "peers" that do some action.

So XVSM provides (some parts are not implemented in the current release) to persist the data, spread it over multiple peers using replication techniques, perform requests with different priorities etc. This can help to easily implement scenarios that perform load-balancing in a path of executions, and/or support recovery by re-starting the peer and retrieving the data from the persistent storage. You can also be notified if certain data has arrived or was taken, or about execution of any other action in your execution path. With these properties, you can easily create a grid alike computing framework where the calculating entities can dynamically join or leave the network.

One main property of XVSM is that it is extensible: you can dynamically improve the functionality of your peers during runtime.