Distributed Services in a Messenger Environment[1]

Murhimanya Muhugusa   Giovanna Di Marzo
Christian Tschudin[2]
Jürgen Harms
Centre Universitaire d'Informatique, University of Geneva
24, rue Général Dufour, CH-1211 Genève 4
Phone: +41 22 705 76 43, Fax: +41 22 705 77 80
e-mail: muhugusa@cui.unige.ch

Cahier du CUI n° 105

November 15, 1996

[1]This work is supported by the Swiss National Fund for Scientific Research (FN-SRS) grant 20-40631.94

[2]Institute fur Informatik, University of Zurich

Abstract

Mobile code is more and more accepted as a promising alternative for structuring distributed applications. The messenger paradigm is the approach to mobile code used in this paper. Messengers are threads of control that can move in a network of messenger platforms to search for resources and information they need to accomplish their task. They are expressed in the **MØ** *messenger language and are interpreted by messenger platforms.* **MØ** *is one such platform. We present in this paper an architecture for distributed services in a messenger environment, i.e., for services implemented with messengers and that are designed to run in a network of messenger platforms. The interaction between a service and its consummers (for example between a server and a client) requires that the interaction interface be known by the two parties before interaction can occur. The classical approach is to establish this interface at development time, making it quite impossible for a client to interact with a service for which it has not been prepared for. With the combined potential of mobile code and code interpretation, it is no longer necessary to know the service interface at development time in order to implement a client that can interact appropriately with the service. In fact, the service is published with its operational interface which describes the necessary information to interact with it. The client discovers the service at run time time, interprets its interface description and collects from it at that moment the necessary information to interact with the service. A service interface is described in an interface description language that is understood by the service clients. Moreover, the use of mobile code and the ability to generate and interpret code on-the-fly make it useless to predefine a protocol in order to achieve interaction between a service and its client. The service can unilateraly move code on the client side where it is executed.*
Keywords: distributed service, messenger, messenger environment, interface description language, **MØ**.

Chapter 1

Introduction

Most Distributed applications are implemented using the server/client [?] programming model. In this programming model, the client sends requests which are executed by the server and receives back from the server the results of its requests. The key point in this approach is that the client must know exactly how to communicate with the server. This means that a non ambiguous protocol must be established between the server and the client before they can exchange information. As far as the user is concerned, the server is seen as a repository of information with which communication must occur in order to collect the appropriate information to achieve a given service.

As an example of a client/server based application, let us consider an application that uses a Web server [?] to access information disseminated in the Internet and to display it on a user screen. For interacting with the Web server, the application will need to implement the HTTP protocol [BLFN96] in order to exchange information with the Web server and to understand the HTML language [BLC95] in order to display data received from the Web server. In order to use what can be called here, the *Web service,* a very complex interaction must be handled between the client and the server (HTTP protocol and HTML language). Moreover, the client is required to participate explicitely is this interaction. What is undesirable in this approach is that changes to the HTTP protocol or the HTML language can require the application to be changed. The reason is that, what is seen as the server does only part of its job relying on the client to accomplish the other part of the job. The client is tightly coupled to the server, and the server imposes much burden on the client. In the case of our example, this translates in the fact that the client must know both the HTTP protocol and the HTML language.

What we propose is to remove the burden imposed by the server on the client by having *true services.* Instead of having a client on one part, and a server on the other part, we will have a service consumer or service user and a service provider. The consumer and the provider of a service should be linked as loosely as possible so that any change in the implementation of the service does not affect the behavior of the service consumer. Therefore we must realize a shift of

functionality from what is seen as the client in the client/server programming model to what is seen as the server in the same model. Returning to our Web example, we would like the user of a Web service to be completely unware of how data reaches its host and how it is displayed on the screen. This means that the service user would not need to know neither the HTTP protocol nor the HTML language. The user has to know only how to invoke the Web service to display a given document located in the Internet. The real job of bringing the document on the user's host and displaying it should be done completely by the server. This notion of service conforms well to what a user can expect from a service provider.

The key concepts for realizing *true services* as described above are code mobility and code interpretation. Our starting point is the thesis developed by Christian Tschudin in his Ph. D. dissertation [Tsc93]. This can be stated informaly as follows: *computer communication based on the exchange of programs is an instructional process which can be used to realize communication between two hosts without a previously established protocol between them. The initiator of the communication chooses unilateraly the rules to use to achieve data exchange, the other part learns how to use these rules.* The possibility of realizing computer communication between entities which do not share the same protocol, together with the ability to create and execute code on-the-fly using code interpretation are exploited to realize *true services.*

In our approach, the provider of a service chooses unilaterally the implementation of the service independently of the implementation of potential consumers of the service. The service provider does not expect the service user to participate to accomplish the service by doing part of the job. This removes the necessity of a complex interaction between the provider and the user of the service. All the user of the service has to do is (a) learn how to invoke the service and (b) invoke the service in the appropriate way.

Now that we know the key concepts for realizing true services, let us see how a Web service could be implemented. As a result of a mouse click on a link in a HTML page, a messenger could be generated. This messenger could locate the Web service and invoke it with the appropriate information. The job of the service user would be as simple as that. Once the Web service is invoked, it would create a messenger which would room in the Internet, probabily using the service of other messengers, to find the information requested by the user. Then a new messenger would come back with the information on the user's host and take the appropriate actions to display the information on the user's screen. This could involve correct interpretation of the data format and probabily the invocation of the proper service to display the information, for example invoking a service for displaying PostScript documents if the received data was a PostScript document. As a last action, the user of the service could be informed of the completion of the service.

The main advantage of this approach is greater flexibility. There is no more any need to define and standardize protocols to suit any kind of applications.

Chapter 2

Architecture for Distributed Services in a Messenger Environment

Messengers [Tsc93] are mobile threads of execution useful for structuring distributed algorithms [DMMTH95]. The execution of a messenger takes place in a *messenger platform*. One such platform is the `MØ` platform [Tsc94]. Several messenger platforms are connected by unreliable channels through which messengers are sent as simple data packets.

Inside a given platform, messengers are *sequential processes* executing *concurrently and in parallel*. They *coordinate their execution* by the means of (1) a shared *dictionary*, and (2) *messenger queues*. The dictionary is a shared data structure accessible to all messengers. A messenger can insert new data, change or remove old data in/from the dictionary. A messenger is able to insert itself in a queue, its executionis then stopped until it reaches the head of the queue. Messengers can *create at run-time* (1) new data, (2) new messengers in the platform where they are executing, and they can *move* themselves or *send other messengers to other platforms*. A distributed messenger environment is a set of messenger platforms linked through unreliable channels and through which messengers can move from platform to platform.

To summarize, messengers are anonymous and autonomous sequential processes, executing in parallel and able to move between platforms and to coordinate theirwork by the means of shared data structures and messenger queues.

In a distributed messenger environment, each platform offers only local services. Inter-node services, i.e., services which require the cooperation of different platforms, are implemented by messengers by letting messengers move through the network to look for the appropriate resources, data or information in order to accomplish their task. In a messenger system, it is likely that different platforms will offer different services. Moreover messengers will enhance platforms where they are running with different services. Hence a messenger reaching a given

platform must be able to locate resources, data and services available in the platform and use them appropriately. For this we propose a service architecture based on two sets of conventions:

1. Common conventions shared by all messenger platforms (called the P conventions below) for locating and accessing the resources available in a platform;
2. Uniform conventions shared by all messengers (called the M conventions below), i.e., services on one side and service consumers on the other side, for publishing, discovering and using services.

The number and the nature of conventions agreed upon in each of the two sets will determine how families of messengers offering different services and those using the services can interact.

The idea here is to adopt a flexible and dynamic approach. The service provider determines how its service can be used, and the consumers of the service learn at run time how to use the service. According to situations prevealing at run time, a service provider can dynamically change the policy enforced to use the service.

We present in the following sections (a) how distributed services can be published in a secure way by service providers, and discovered by messengers which need them to accomplish their tasks, (b) how service consumers can learn the policy enforced by the service provider to allow them to use the service in an appropriate way and (c) how to deal with the dynamicity induced by hosts joining the system and those leaving it.

2.1 Service Publication and Discovery

In a messenger platform, messengers exchange data only through a global store. Data is stored in the global store as a pair consisting of a key that we will call the *data key* and an associated value that we will call the *data value*. We will say that data is stored in a global store under the data key. Normally, this key is also used to access the data. It is the responsibility of a data creator to publish the data key to messengers which are intended to access the data it has created.

To allow the publication of services, the following convention must be shared by all the platforms of the messenger environment.

Convention P 2.1 *The global store provided by each messenger platform is sub-divided (structured) in two separate functional areas: (a) the* global dictionary *used for data exchange between messengers and (b) the* service dictionary *reserved for the publication of services. These two dictionaries are named in a uniform way by all the platforms, they are all read-write accessible to all messengers. The global dictionary is not a browsable dictionary, while the service dictionary is a browsable dictionary.*

The global dictionary is not browsable while the service dictionary is browsable means that, to access data stored in the global dictionary, one must provide the exact data key under which the data is stored in the dictionary. For the service dictionary, it is always possible to determine the exact key under which a data item is stored in the dictionary because any messenger can browse the dictionary and construct the list of data values contained in the dictionary, together with their associated keys. Hence the difference between the two dictionaries is that data stored in the service dictionary can be "seen" by all messengers while data stored in the global dictionary is only "seen" by messengers which know the data key, i.e., those messengers that have got the data key from the data creator or from another messenger which knows the data key. As data stored in the service dictionary can be seen by all messengers, this dictionary is intended to be used only for the publication and localization of services.

Convention M 2.1 *A messenger or family of messengers which offers a service usable by other messengers publish the service by adding an entry describing the service in the service dictionary. The service provider is responsible for ensuring that its service is published in a secure way. A messenger reaching a given platform can determine local services available in that platform by browsing the service dictionary.*

Messengers offering services add entries describing their services in the service dictionary. However, since all data stored in this dictionary is seen by all messengers service providers must ensure that their services are published in a secure way, i.e., that unauthorized messengers cannot wipe out willing-fully or accidentally entries they have stored in the service dictionary.

Given the above two conventions, a service provider can publish a service in a secure way as follows:

1. The service provider creates a private data structure and fills it with the appropriate information to fully describe the service being published. This data structure can be considered as the *service access point (SAP)* for the service being published.[1] Information stored in this data structure is determined by the service provider and provides the service consumer hints and various instructions on how to use the service. Section 2.2 gives more information on this topic.

2. The service provider chooses a public service name that will be used by service consumers to identify the service. A service consumer must know somehow the public service name of a service it wants to use.

3. The service provider chooses a secret key and generates from it a public key using a one way function.[2]

[1] In `MØ`, this data structure will be most of the time a dictionary.

[2] The `MØ` platform provides the necessary operators for encoding/decoding keys. They are based on DES.

4. The service provider adds in the global dictionary an association between the public key generated in step 3 as the data key and the data structure created in step 1 as the data value. The service provider secures the created association with the secret key chosen in step 3, i.e., the association can only be removed from the dictionary by messengers which know the secret key used to secure the association. In fact, each platform provides an operator which atomically creates an association in a dictionary and secures it with a given secret key.[3]

5. The service provider adds an association in the service dictionary between the public key generated in step 3 as the data key and the public name chosen in step 2 as the data value. As in step 4, the service provider secures the association added in the service dictionary with the secret key chosen in step 3. Because the service dictionary is browsable, the association is made, at this stage, visible to all messengers but it can be removed from the service dictionary only by messengers which know the secret key used to secure it. Now, the same public key appears both in the service dictionary where it is associated with the public service name and in the global dictionary where it is associated with the service access point.

If the secret key generated in step 3 is kept secret, the above procedure allows a service provider to publish a service without having other messengers remove any trace of the service. Moreover, messengers reaching the platform can locate the service, provided they adhere to the two conventions given above.

In fact, a messenger that needs to use the service, must know the service name that identifies the desired service. Then the messenger locates the service by browsing the service dictionary in the platform. It finds the service name used to identify the service and learns the public key associated with it. This public key is then used by the service consumer to access the service through the service access point (the data structure created in step 1) stored in global dictionary.

As an example, figure 2.1 shows how a distributed semaphore service is published in the **M∅** platform. Firstly the dictionary **semadict** is created with read/execute access only to other messengers and is initialized with information describing the distributed semaphore service. Such information contains for example, the **P** and **V** procedures used to acquire and release a semaphore. The **semadict** dictionary is added in global dictionary, called **globdict** in **M∅**, with the **secretdef** operator using the secret key **k1** as a parameter. The **secretdef** operator atomically creates the public key **k2** and adds the association (**k2**, **semadict**), secured by the secret key **k1**, in **globdict**. Secondly,

[3]The **M∅** platform provides the **secretdef** operator for that purpose. This operator takes three arguments: (a) the dictionary where the association will be added (b) the data value and (c) the secret key. It atomically generates a public key and adds in the given dictionary an association between this public key and the data value. The association is secured because it can only be removed from the dictionary by the **secretundef** operator which requires as a parameter, the secret key used to create the secured association. Moreover, the association can be altered only by the **secretdef** operator with the same secret key.

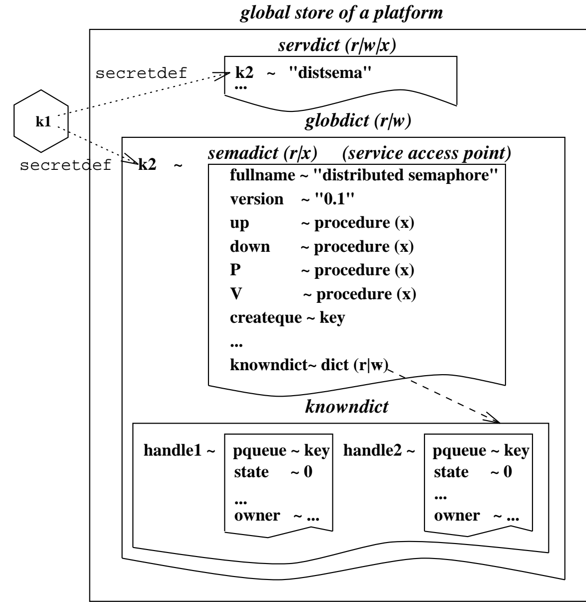

Figure 2.1: Publication of a service in the **M∅** platform.

the string **"distsema"**, chosen as the public name to identify the distributed semaphore service, is **secretdef**ed in the service dictionary, called **servdict** in **M∅**, using the same secret key **k1**. As above, this results in the association (**k2**, **"distsema"**) being added in **servdict** and secured by secret key **k1**. Now, the distributed semaphore service is accessible to other messengers. The user of this service will browse the **servdict** dictionary searching for the **"distsema"** string. It will find that the public key **k2** is associated with **"distsema"**. Then it will use the key **k2** to access the distributed semaphore service's SAP (**semadict**) as this SAP is stored in **globdict** under the same key.

2.1.1 Discussion

The drawback of the approach presented here for publishing and discovering services is that the public service name of a service must be known before the service can be used. This means that the service provider and consumers must agree before hand on the name that will identify the service. However, this approach allows a simple but yet flexible way of publishing and discovering distributed services as unlighted by the following facts:

Dynamic addition/removal of services

The contents of the service dictionary are not fixed when a platform is brought up. On the contrary, they dynamically change as messengers add/remove entries

describing services in/from the dictionary. Removing an entry describing a service from the service dictionary can be used to hide the service from messengers which do not yet have a reference on the service's SAP.

Coexistence of different implementations of a service

This approach allows multiple implementations of a service to run concurrently in a platform. For example, one implementation can add or define functionality that is intended to be used only by some messengers. The coexistence of multiple implementations of a service can be achieved in two ways:

- Instead of associating only one SAP to the service, multiple SAPs are defined, each of them corresponding to a different implementation of the service. However, only one of the SAPs is "visible" at any time, the others being "hidden". The service provider can then switch through the different SAPs, i.e., making a different SAP visible, according to its own chosen policy. In this way, messengers that access the service when a given SAP is visible, will use the implementation of the service associated with that SAP. Other messengers will likely access the service when a different SAP is visible and though will use a different implementation of the service. In this case, it is the service provider that unilaterally determines the implementation to be used at any moment, leaving no choice to the service consumer.

- The alternative is to give the service consumer the freedom to choose among the different implementations. This is done by defining multiple entries in the service dictionary under the same public service name. Each of these entries describes a different version of the service. Each entry has a unique public key, and therefore can have its own SAP, i.e., each version of the service will have its own associated SAP. The SAP will contain specific information, e.g., the version number, that will allow to uniquely identify the version of the service to which it is associated.

  This feature can also be used to allow competing implementations of a given service to run concurrently on a platform. One can imagine that different service providers offer the same service, and therefore, use the same service name to identify the service. The service consumer will choose one implementation to use according to its own criteria.

Because all services should be published and located through the service dictionary, this dictionary should be big enough to store all the entries added by service providers. Moreover, entries describing services are secured in the service dictionary by secret keys chosen by service providers. Therefore, if two service providers choose the same secret key to publish different services, confusion will certainly arise as either of the providers can erase or overwrite information added in the service dictionary by the other. This will result in the failure of either one of the services or all of them. As the secret key is also used to generate

the public key under which the service's SAP is stored in the global dictionary, a clash name in secret key will result also in a clash name in public key.

Our approach to this problem is a probabilistic one. By letting each service provider choose a random secret key of enough length (for example a random key generated before hand by the system), we can reduce the probability of name clashes.[4]

Since the service dictionary is write-accessible to every messenger, a messenger with "bad intentions" can try to fill it with dummy entries so that other messengers cannot publish their services. This is an attack for denying other messengers write access to the service dictionary. Service denial is an attack that attempts to exploit some weakness in the management of common resources. Therefore, the messenger platform should be responsible to combat such kinds of attacks because the platform is responsible for managing available common resources. A solution to this problem would be to have the messenger platform charge messengers for entries they add in the service dictionary,[5] and adjust the price of adding entries in service dictionary according to the number of entries that a messenger already has in the dictionary.

Another possible attack in an open environment is to have a messenger add dummy entries in the service dictionary using the public service name of a service published by another messenger. Here the service consumers get confused as it becomes difficult for them to distinguish among the visible services, the right service to use. It is the responsibility of the service provider to devise a policy to protect itself against such attacks. One approach that a service provider could use, would be to provide a procedure implementing a kind of "test-and-challenge" that the service consumer can execute to ensure it is talking to right service provider. Here again, the service provider and consumer must agree somehow before hand, on some "shared secret", i.e., on the interpretation of the results of the "test-and-challenge" procedure.

Classically, the publication of services is not done through a shared "resource" accessible to every process. On the contrary, a solution based on a kind of "name server" is usually used. The name server protects the shared resource used for publishing services by hiding it to other processes. Therefore, service providers must register their services through the name server. The name server generates a name which uniquely identify a registered service. Although this

---

[4] In M0, keys generated by the system are random strings of 64 bits. Given a service dictionary containing 1000000 entries, and given that all the keys are generated randomly by the system, the probability that the next entry will use the same key as one of the available entries is $1000000/2^{64} < 6 * 10^{-14}$. Although this does not ensure that an name clash cannot arise, we can consider that this is impossible with a great confidence. In fact, the probability that the system fails due to other bugs will certainly be more higher than that of a name clash.

[5] In M0, messengers pay for the different resources they consume for their execution: CPU time, memory and network bandwidth. The platform can have more control on resource consumption by dynamically adjusting the costs of the different resources according to the prevealing situation. If a messenger tries to monopolize portions of memory shared by other messengers, the system can raise the cost of that portion of memory. This is similar to taxes introduced by same states to combat any attempts from service providers to establish monopoly or to create trusts.

approach ensures that name clashes cannot occur, it presents some drawbacks:

- It requires the development and implementation of a complex protocol for generating unique names across a network of hosts. The more complex the protocol, the more likely its implementation will be error prone. This can also restrict system flexibility if a change in the protocol requires that other components of the system have to be changed. The impact of a change in the protocol can be reduced if the implementation is done through an external server, provided that basic components of the system do not rely on that protocol.

- Usually, the name server is implemented as a process or a set of cooperating processes. Therefore a failure of the name server will result in the failure of other services, probably also services already running in the system.

- A service provider cannot choose a name for its service. The service name will be known only at runtime and depending on the implementation, the service name can change on each execution of the name server (for example after a reboot).[6] As a consequence, a service provider cannot "advertise" its service as it does not have any control on the information that identify its service.

- Classical implementations of name servers do not provide simple ways of handling concurrent competing services.

- Further structuring of the service dictionary to allow scaling (structure the service dictionary into domain dictionaries)
- Some words about service semantics and high level methods for describing service semantics

Based on two simple conventions, one to be shared by all the platforms of a messenger system and the other by all messengers, we have proposed a simple and flexible way for publishing services. It allows service providers to publish their services in a secure way and ensures that all services published are visible to all messengers. Each messenger has free access to the common dictionary used for the publication of services and has freedom to choose a name to identify its service. This freedom allows different configurations such as the coexistence of competing services, that are not normally offered by classical approaches. No single protocol is enforced between the different hosts to achieve publication and discovery of services, only local resources available in a platform are needed for that. Service providers are responsible for protecting their services in an open environment where some messengers can behave badly. The platform offers the basic mechanism necessary for that. Beside making a service visible to other messengers, a service provider must provide the service consumer the necessary

[6]In the UNIX system, ports are used to identify services. A range of ports is managed by an authority which ensures that registered services will always be assigned the same port. Other services can freely choose a port in the range not managed by the official authority.

information to use the service appropriately. The information describing a given service is accessible through the service's SAP, however, it must be interpreted in a consistent way by both the service provider and the service consumer. The next section presents how this can be done.

2.2 Service Interface

Once a messenger has found the right service to use in order to achieve its task, or to proceed in the completion of its task, it is confronted to another complex problem, that of using the service in a coherent way. As an example, let us consider a distributed semaphore service which allows different messengers to synchronize their execution independently of their physical location. Messengers which intend to use this service must somehow know the different steps they have to follow: (a) create a semaphore and (b) use the semaphore for synchronization. Moreover, they must know how to create the semaphore and how to use it, i.e., how to acquire a semaphore and how to release it. Service consumers must exactly know the required protocol to interface with the service.

How can a service consumer know the protocol required to interface with a service? This is not an easy problem. The commonly used solution is to establish a protocol before hand between the provider and the consumer of the service. The protocol defines (a) rules to follow when accessing the service and (b) clear semantics for interpreting different events which can occur in the interaction between the provider and the consumer of the service.[7] The protocol is then embedded in the code of both the provider and the consumer of the service. This means that the provider and the consumer of the service are statically configured so that an interaction can occur between them.

Although this static configuration approach seems simple, it lacks flexibility. As a consequence of this, an application developer must know at development time, how to interface his/her application with all services the application will require to accomplish its task because the application must contain the various protocols to interface with those services. A change in the protocol to interface with a service will surely require that both the provider and the user of the service be changed. There is no simple way for an application to use a service for which the interface is not known at development time.

2.2.1 The Client/Server approach

In most implementations of distributed applications based on the client/server programming model, service providers and service consumers are implemented as independent processes which communicate by exchanging messages according to the established protocol. The client application must at least conform to a

[7] The protocols to interface with basic and other useful services are standardized to allow an interaction between providers on one hand and consumers on the other hand, implemented by different people or organizations.

protocol defining rules for *horizontal interaction* (also called horizontal protocol in this text), i.e., for interaction with the server process probably through a network. Most of the time, the client application contains also a protocol defining rules for *vertical interaction* (also called vertical protocol in this text), i.e., for interaction between the different parts of the client. In fact the client is usually structured in a part responsible for interaction with the server (the network part of the application), and other parts which do the real job, using data and information received from the server through the network part of the application (see figure 2.2). The reason for this structuring of the client in a network part, and other parts, is precisely to confine to a single module of the client application "what depends on the server", i.e., what has to be changed if the protocol for interaction with the server is modified.

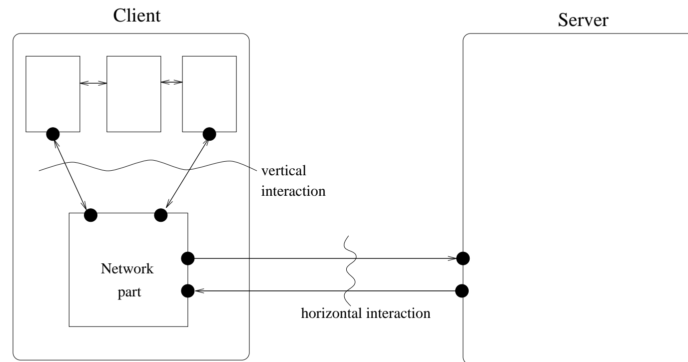

Figure 2.2: Different interaction protocols for a client application.

Two examples will help us understand the difference between horizontal protocol and vertical protocol.

1. Our first example is that of transferring files between hosts. The user requesting the transfer of files runs on his local host a client process which will establish a connection with a server running on the remote host using the FTP protocol [PR85]. On one side, the client communicates with the server using the FTP protocol: this is the protocol for horizontal interaction. On the other side, the client process interacts with the user through a kind of restricted command interpreter. The client receives from the user commands such as **`get filename`** or **`put filename`** and generates from them appropriate sequences of requests in the FTP protocol for execution by the server. Here, the restricted command interpreter implements the protocol for vertical interaction.

2. Our second example is again an application using the Web. A user uses a Web browser such as the NCSA Mosaic [] or Netscape [] to navigate into HTML documents disseminated into the Internet. Whenever the user requests the display of a document through a mouse click on a link, the Web

browser creates the appropriate request and sends it to the appropriate Web server using the HTTP protocol. The server will send back a result; a HTML page that will be displayed by the Web browser. Here the HTTP protocol and the HTML languages constitute the protocol for horizontal interaction. The user display, the mouse events and the HTML languages are part of the protocol for vertical interaction (interaction between the client and the user).

What appears from these examples is that, usually the protocol for interaction between the client and the server is rather complex as it must define among other things, formats of exchanged messages, the semantics of each type of message and the reaction of both the client and the server on each type of message. Due to the complexity of the horizontal protocol and the fact that this protocol is seldom defined by the client developer, the implementation of the network part of the client is error prone. On the contrary, the protocol for vertical interaction is completely defined by the client developer and is usually simpler than its counter part for horizontal interaction. The protocol for vertical interaction essentially defines (a) how interaction occurs between the network part of the client and the other parts and (b) the nature of information exchanged between the network part of the application and the other parts (e.g., parameter types for calling service routines in the network part, the sequence of calls to follow for calling service routines, how to signal asynchronous events through signals or triggers, etc).

2.2.2 The RPC approach

The implementation of a distributed application using the client/server model can be simplified by using the RPC paradigm [Nel81, Blo92]. Under the RPC paradigm, the client "sees" its environment enhanced by procedures which actually reside inside the server. A layer of RPC software is added in both the client and the server hosts and implements the protocol for horizontal interaction between the client and the server. The protocol for horizontal interaction is shifted out of both the client and the server. This means that neither the client, nor the server has to know the protocol for horizontal interaction as it was the case when RPC was not used. The server appears to the client as a set of *local procedures* and therefore the interaction between the client and the server is restricted to procedure calls. As far as the client is concerned, only a protocol for vertical interaction is needed to interact with the server. In fact, the network part of the client is replaced with the RPC software which has to interact with the other parts of the client using the protocol for vertical interaction. Similarly, the client appears to the server as a process running on the same host and which can request the execution of some procedures (see figure 2.3).

Despite the complexity added by the RPC software layer, the RPC paradigm surely brings more flexibility as it suppresses the need of a protocol for horizontal interaction between the client and the server. However, both the client and the server are constrained to use the same protocol for vertical interaction with the

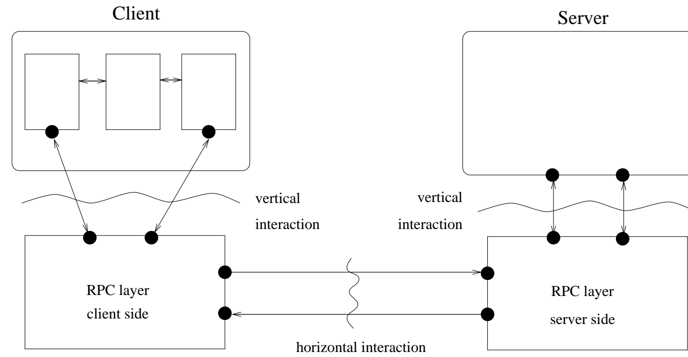

Figure 2.3: Interaction between client and server using RPC

RPC software layer. In the case of RPC, the vertical protocol defines (a) the names of procedures located in the server address space and callable by the client and (b) the type of the arguments required for calling those procedures. While in the approach without RPC the horizontal protocol is preestablished before hand and the vertical protocol is chosen by the client, with RPC, the client and the server must agree on the vertical protocol before hand. Once again, this results in a configuration rather static, i.e., not completely flexible.

2.2.3 A more flexible approach

We propose here an architecture for the implementation of distributed services in a messenger environment which does not require that the provider and the consumer of a service agree before hand on any kind of interaction protocol. The idea is to fully exploit the combined potential of mobile code and interpreted code which makes possible the dynamic configuration of protocols at run time. This is achieved by letting the service provider choose unilaterally both the protocols for vertical and horizontal interaction and by letting the service consumer discover (learn) how to interact with the service at run time.

More precisely, this is done by moving code from the service provider's host to execute on the consumer's host. This code is part of the service provider that will instruct the service consumer's host how to interact with the other part of the service provider executing on the provider host. This code will also carry out the necessary actions to accomplish the service requested by the service consumer. This means that the code sent by the service provider for execution on the consumer's host will contain both the appropriate protocol for horizontal interaction with the service provider —the other part of the service provider— and the vertical protocol for interaction with the consumer application. Neither the consumer nor the consumer host need to know the protocol chosen by the service consumer for horizontal interaction; i.e., for interaction between code sent by server for execution on the consumer host and the other part of the service provider. In order to accomplish the service requested by the consumer

application, code sent by the server on the consumer's host can interact with the consumer application through the basic interaction mechanisms available to messengers: global memory and process queues. Our approach is to let the service provider choose unilateraly how this interaction should occur and to let the client discover how to accomplish this interaction. The service consumer learns how to interact with service through the service interface which contains all the necessary information to use appropriately the service.

There are some similarities between our approach and RPC: (a) the service consumer does not need to know the protocol for horizontal interaction with the service provider as this interaction is accomplished by code external to the service consumer; and (b) only vertical interaction is necessary to interact with a service. However in RPC, (a) the vertical protocol is preestablished and is restricted to procedure names and their parameter types and (b) code (procedures) that handles the client requests is executed on the server host and does not share any address space with the client application. On the contrary in our approach, the vertical protocol is not preestablished and part of the code that accomplishes the consumer service executes on the same platform as the consumer application and therefore shares an address space with it.[8] A consequence of having a shared address space between the provider and the consumer of the service is that (a) there is any more no need of data marshaling and (b) the possibility to exchange any kind of complex data structures between the consumer and the provider of a service.

To sum up, we propose a flexible approach where the provider and the consumer of a service interact without a preestablished protocol. The service provider provides "objects" and parts of its code that can be directly accessed or executed by the service consumer in a controlled way. The consumer learns how to use the service —to interact with the server and other messengers using the same service concurrently— in a coherent way. The necessary information for using the service is provided by the service provider in the service SAP. The service SAP contains the service interface description expressed in a common language shared by all messengers. The next section describes this language.

Describing a Service Interface

The service SAP is stored by the service provider in the global dictionary and contains the necessary information to allow the service consumer to use the service in a consistent and appropriate way. The service SAP is located by the service consumer as described in section 2.1. The service SAP must imperatively contain a description of the service interface. Of course, it can contain any additional information the service provider wants to publish. More precisely, all the messengers must adhere to the following convention:

[8] There is another difference because the RPC software must be preconfigured in both the client and server hosts before RPC can be used by applications. This means that a protocol has to be preestablished between the two parts of the RPC software. Therefore, preconfiguration is used not only at the application level but also at the lower level that supports the execution of applications. In our approach, only a platform for the exchange and execution of messengers is needed in each host.

Convention M 2.2 *Each provider of a service must provide a complete description of the service interface in the service SAP. Before using a service, any messenger should learn how to interact with the service by looking for the appropriate information in the service SAP. The provider of a service uses the messenger service interface description language to describe its service interface to ensure a common interpretation of the interface by both the provider and the consumer of the service.*

The preferred way to interact with a service is through a functional or procedural interface. In fact a procedure call has a simple and non ambiguous semantics. When a procedural interface is used, the service consumer sees the service as a black box hiding the details and complexity of actions which accomplish the service. And as far as the service provider is concerned, a procedural interface is a simple way to grant access to "resources" in a contolled way. The user is not required to know the policy for using the resource, it only calls a procedure which enforces, behind the scene, the policy chosen by the service provider.

Although a service consumer interacts with a service through a procedural interface, the actual implementation is chosen by the service provider. The implementation can involve the creation of a family of messengers which spawn the network of platforms looking for the necessary information or resources to accomplish the service, and which coordinate their execution, probably also with the consumer of the service. This means that most of the time, the service consumer is invited to "spawn" another messenger which will execute the procedure as part of its code.

Before looking at the proposed messenger service interface description language, we present the necessary information to describe a procedural service interface in a messenger environment. A description of an interface should contain the information on:

1. Procedures and their parameters: the service provider must give a description of all the procedures the service consumer can call to interact with the service. The description of a procedure must give the service consumer the information for locating and for using the procedure. For calling a service procedure, the service consumer must know what parameters the procedure expects, what parameters are modified and how to get back the results of the procedure execution. Therefore, a procedure description must explicitely state (a) what parameter types are expected and (b) what parameters are modified.

   Describing procedures and their parameters in a messenger environment seems quite simple because all messengers are expressed in the same messenger language. Therefore, procedure parameter types will be limited to the "object types" provided by the underlying messenger language. However, the provider of the service can give the service consumer more hints on the parameter semantics so that the consumer can supply the right

information to the procedure.[9] We have found useful to express clearly the following attributes for some parameters in a messenger environment:

(a) origin: most of the time, the service provider will need to know the address of the host where the consumer is executing. When this information is needed as a parameter to a procedure, the service consumer must know that it must provide the address of its origin. In this case, it is not sufficient to state that a parameter of type array for example is expected (provided that addresses are expressed as arrays).

(b) platform identifier: as for the origin information, the service provider must state explicitely when a procedure expects the identification of the platform where the consumer is executing. This information is different from the origin. Indeed, a single host can contain different messenger execution platforms, each identified by its own platform identifier. All these platforms will share the same origin information.

(c) channel: to send a messenger back to the consumer's host, the provider of the service can need to know the appropriate channel to reach the consumer's host. If the provider states that a "channel" parameter is expected, the service consumer can provide the right information.

(d) secret key: the execution of some procedures can need to be protected somewhow; i.e., the provider of the service must determine that the consumer is allowed to execute that procedure. One way to check this information, is to have the consumer provide a secret key received before hand and granting it the ability to execute the procedure.

(e) interaction queue: when a procedure parameter is given this semantics, the caller of the procedure knows that it must take some special actions. An interaction queue is used by the procedure to signal the caller that the procedure has completed its execution. More precisely, the caller of the procedure is expected to wait for the result of the procedure in the interaction queue; at procedure completion, the caller is awakened so that it can continue its execution.

(f) internal or private: some services create internal objects that are afterwards referred to by "handles". When a parameter is given *internal semantics*, the caller of the procedure is expected to provide a *handle* created before hand by the service provider and which is used in a special way by the service provider to identify objects on which to act.

    As an example, one of the messengers using the distributed semaphore service can request the creation of a semaphore to coordinate their

[9] A different approach would be to use a more complex language that allows to describe completely also the semantics of the service, the different procedures and their parameters. This approach could allow a messenger to learn the semantics of an unknown service and to use the service afterwards. In our approach, the user of the service is supposed to know the semantics of the service; it learns only how to interact with (use) the service.

execution. The service provider can create an internal data structure representing such as semaphore and give to the caller only a handle to identify the real semaphore. By giving to the caller only a handle to identify the real object, the service provider hides the object and protects its integrity. All the other procedures which act on the semaphore will require that the user provides the handle to identify the semaphore.

2. Admitted sequence of procedure calls: most likely, the interface of a service will contain the description of more than one procedure. In this case, the description of procedures only might not be sufficient for an appropriate usage of the service. When the procedures must be called in a predefined order to ensure the consistency of the service or to be meaningful, the service provider must include in the service interface description, the relevant information to allow the consumer to call the different procedures in the right sequence.

   Let us use again the distributed semaphore service as an example to illustrate this point. Before a semaphore can be used by messengers to coordinate their execution, one of the messengers must create the semaphore. Afterwards, the procedures for acquiring and releasing the semaphore can be called when needed. In this case, the service provider must state clearly that the procedure for creating the semaphore has to be called first. And once a semaphore has been created, the other procedures for handling it can be called in any sequence.

3. Expected interaction between the provider and the consumer of the service: as messengers can move form host to host in a messenger environment, it not always clear where the interaction between the provider and the consumer of the service has to occur. In fact this can occur either on the provider or on the consumer host. Therefore, when the consumer is required to provide, for example an interaction queue as a parameter to a procedure, the consumer has to know in which host the queue is expected to be located by the provider of the service.

4. Service interface type: a service interface can be either static or dynamic. For some classes of services, once a user of a service has learned how to use the service, the interface is expected to remain unchanged (stable). The service interface is a static interface. In this case, the user can acquire once, references to the procedures it needs to call. All subsequent procedure calls will use these references. However some service implementations can require a dynamic interface. In this case, the user is expected to check if the interface has changed whenever it needs to interact with the service. This can imply that the user go again through the learning process if the interface has changed. A dynamic interface is attractive only if the interface changes very often and if the changes are not confined only to a limited part of the interface. On the contrary, if the service interface changes rarely, and if changes do not affect a large part of the interface,

the report of asynchronous events described below should be used as an alternative to the dynamic interface.

5. Report of asynchronous events: a service provider can need to report some asynchronous events to the users of its service. Some examples of such asynchronous events are the change of the service interface and the shut down of the service. Indeed, a complex service can rely on a number of more basic services provided by other messengers. If one of the basic services fails, the service provider can decide to shut down its own service and reports this event to all the users of the service.

2.3 Interface Description Language

We propose here a "language" called *messenger interface description language* that can be used to describe the information contained in the service access point (SAP) of a given service. A service's SAP contains all the necessary information for interacting with the service. Figure 2.4 shows the structure of a service access point.

A service's SAP is a dictionary that contains a varying number of entries (information) describing the SAP together with the service that is accessed through it. A service's SAP contains at least three mandatory entries, the other entries are optional. The mandatory entries are:

- the entry under the name `type` that describes the type of the service access point;
- the entry under the name `interface` that describes the interface provided to messengers for interacting with the service;
- and the entry under the name `sequence` that describes the admitted sequence of procedure calls for interacting with the service.

Rules 2.1–2.22 define more formally a service access point.[10] A service's

[10] Each entry in a dictionary is defined by its *key* and its *value*. Both the key and the value of an entry can be any valid messenger object. For this reason, the required object type is specified after the keywords `key` and `value`. For example, in rule 2.1, the key of a service's SAP is an object of type `name`, i.e., a name, while its value is an object of type `dict`, i.e., a dictionary.

Dictionaries and arrays are complex objects that are further described. A dictionary is described by describing each of its *entries*. These are identified by the keyword `entries`. Some of a dictionary's entries are *mandatory* entries, in which case they must appear in the dictionary. They are introduced by the keyword `mandatory`. The other entries of a dictionary can be *optional* entries introduced by the keyword `optional`. The keyword `many` is used to specify that a dictionary must contain at least one of the entries to which the keyword is applied.

An array is described by describing its *elements* introduced with the keyword `elements`. Contrary to a dictionary, an array defines an order on its elements. The keyword `first` is used to refer to the first element of an array and the keyword `other` to refer to elements not refered to yet.

Finally, the keyword `ANY` is used to refer to any valid object of the appropriate type. For example, `int(ANY)` refers to any valid integer.

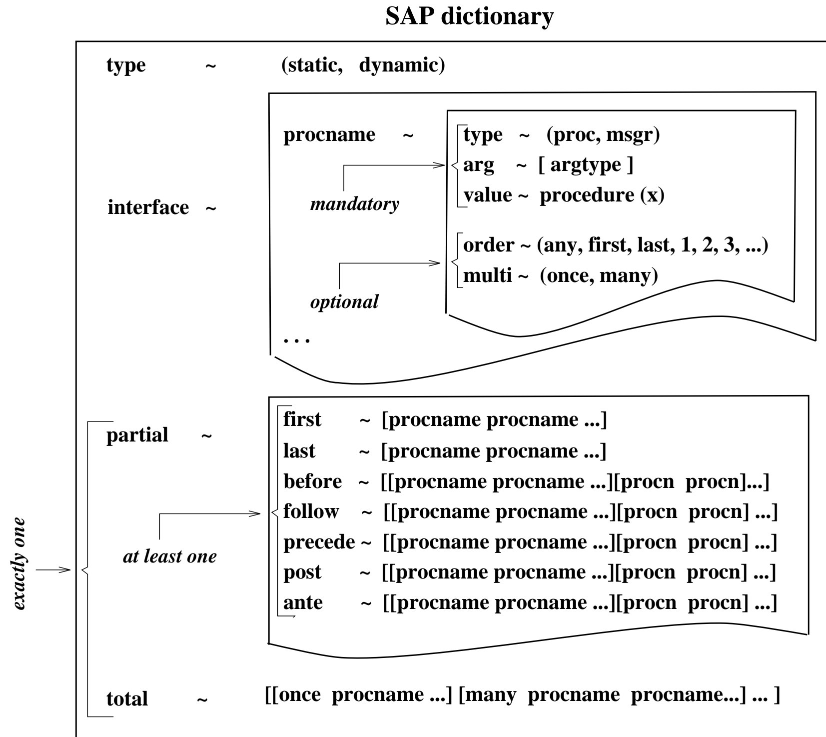

Figure 2.4: Structure of a Service Access Point

SAP is identified by its name and its value which is a dictionary (rule 2.1). The different mandatory entries in the SAP dictionary are further described by rules 2.2–2.11.[11]

$$
\begin{aligned}
SAP \stackrel{def}{\Longrightarrow}\ & \mathbf{key}(name(ANY)); \\
& \mathbf{value}(dict \\
& \quad : \mathbf{entries}(\\
& \qquad \mathbf{mandatory}(Stype, Sinterface, Ssequence), \\
& \qquad \mathbf{optional}(ANY)))
\end{aligned} \tag{2.1}
$$

2.3.1 The Service Access Point's Type

The SAP's type is described under the entry `type` in the service's SAP dictionary (see rule 2.2). Its value can be either `static` for a *static SAP* or `dynamic` for a

[11] Optional entries in the SAP dictionary are not described here. They can be any valid objects provided by the messenger language. The service can store in the optional part of the SAP, all additional information required for the operation or management of the service but not needed by messengers in order to interact appropriately with the service.

dynamic SAP.

$$Stype \overset{def}{\Longrightarrow} \textbf{key}(name(\text{“}type\text{”})); \quad \textbf{value}(name(\text{“}static\text{”} \mid \text{“}dynamic\text{”})) \qquad (2.2)$$

A *static SAP* is used when the interface for interacting with the service is not allowed to change during service execution. A messenger that interacts with a service described with this kind of SAP needs to determine only once the required information for an appropriate interaction with the service.

On the contrary, a *dynamic SAP* is used to describe a service for which the interface can change during service execution. With this kind of service, the information used by a messenger for a previous interaction with the service may not be any more appropriate for another interaction with the service. Therefore, a *dynamic SAP* expresses the requirement that a messenger interacting with the service must determine, by rescaning the service interface, the appropriate information for interacting with the service before each interaction can take place. What is allowed to change during service execution is the interface description and consequently, the admitted sequence of procedure calls for interacting appropriately with the service (see below).

2.3.2 The Interface for Interacting with the Service

The interface for interacting with the service is described under the `interface` entry in the service's SAP dictionary. Its formal description is given by rules 2.3–2.10. The interface itself is a dictionary, called the *interface dictionary*. It contains names of procedures and messengers that must be used to interact with the service, associated to their procedure or messenger description (rule 2.3).

$$\begin{aligned} Sinterface &\overset{def}{\Longrightarrow} \textbf{key}(name(\text{“}interface\text{”})); \\ &\qquad \textbf{value}(dict \\ &\qquad\quad : \textbf{entries}(ProcDesc)) && (2.3) \\ ProcDesc &\overset{def}{\Longrightarrow} \textbf{key}(name(ProcName)); \\ &\qquad \textbf{value}(dict \\ &\qquad\quad : \textbf{entries}(\\ &\qquad\qquad \textbf{mandatory}(Ptype, Parg), \\ &\qquad\qquad \textbf{optional}(Porder, Pmulti))) && (2.4) \\ ProcName &\overset{def}{\Longrightarrow} ANY && (2.5) \end{aligned}$$

A procedure or messenger description (rule 2.4) is a dictionary called *procedure description dictionary*, containing all the necessary information messengers might need to interact with the procedure or the messenger, such as the type of parameters expected by the procedure. A procedure description dictionary can contain up to five entries, three of which are mandatory while the remaining

two are optional. The mandatory entries are described by rules 2.6–2.8. They are:

1. The entry under the name `type`: it defines the type of the procedure described in the dictionary. It can take two values: the name `proc` for a procedure or the name `msgr` for a messenger (rule 2.6).

   Interacting with a procedure is quite different than interacting with a messenger. In the former case, the procedure is executed in the context of the calling messenger and information can be exchanged between the caller and the procedure either on the caller's operand stack or through memory. In the latter case, the caller and the callee are distinct messengers executing in different contexts. They can exchange information only through shared memory.

2. The entry under the name `value`: this is a reference to the executable code for the procedure so that it can be called. When the interaction has to occur through a messenger, the `value` entry is the code of the messenger to be created by the user of the service.

3. The entry under the name `arg`: this is an array called *argument array*. It describes the arguments expected by the procedure or the data required for the execution of the messenger. Each member of the argument array describes the corresponding argument or data item; i.e., the first element of the argument array describes the first argument expected by the procedure or the first data item required by the messenger, and so on. An element of the argument array can be (rule 2.8):

   (a) one of the following names of data types provided by the messenger language: `int, time, name, key, array, mqueue, string, dict, page, pmap, channel, cpu, sponsor` or `account`. In this case, a parameter or data item of the specified type is expected;

   (b) the name `waitqueue` (for wait queue) or the name `ackqueue` (for acknowledgement queue): in this case, a process queue is expected by the procedure or the messenger. Both the wait queue and the acknowledgement queue are only used for interaction with the service through a messenger. However, they convoy quite different semantics.

   On one hand, a wait queue is a queue where the caller waits for the completion of the task or service it has requested. More precisely, the caller stops the wait queue, invoques the service and then inserts itself in the queue. When the callee completes its task, it restarts the wait queue and doing so, unblocks the caller.

   On the other hand, an acknowledgement queue is used when there is a need for further interaction between the caller and the callee after the caller has been unblocked. The typical use of the acknowledgement queue is when, as part of its task, the callee must unblock the caller and ensure that the caller has seen the result of the task before

proceeding. An acknowledgement queue is always used in conjonction with a wait queue. Before unblocking the caller waiting in the wait queue, the callee stops the acknowledgement queue. After it has restarted the wait queue, the callee inserts itself in the acknowledgement queue. Then the caller "processes" the result of the task and acknowledges that the result of the service has been "processe" by restarting the acknowledgement queue; this unblocks the callee.[12]

(c) the name `origin`: in this case, the address specifying the origin of the calling messenger is expected;

(d) the name `internal1` or `internal2` etc: in this case, an "opaque" object created by the service as the result of a previous interaction is expected. A service can created opaque objects having different semantics. To distinguish the different objects, the service uses the type names `internal1`, `internal2` etc. For example, the distributed semaphore service could use an integer to represent a semaphore. However, every integer value does not represent a semaphore for the service, only integers created by the service are meaningful for the service. In this case, it is better to use for the service to use the type `internal1` whenever an integer representing a semaphore is needed. In this way, the service user is aware that the service expects not any integer, but only integers that represent semaphores for the service.

$$Ptype \stackrel{def}{\Longrightarrow} \mathbf{key}(name(\text{``}type\text{''})); \quad \mathbf{value}(name(\text{``}proc\text{''} \mid \text{``}msgr\text{''})) \tag{2.6}$$

$$Parg \stackrel{def}{\Longrightarrow} \mathbf{key}(name(\text{``}arg\text{''})); \quad \mathbf{value}(array[] : \mathbf{elements}(Argtype)) \tag{2.7}$$

$$Argtype \stackrel{def}{\Longrightarrow} name(\text{``}int\text{''} \mid \text{``}time\text{''} \mid \text{``}name\text{''} \mid \text{``}key\text{''} \mid \text{``}array\text{''} \mid \text{``}mqueue\text{''} \mid \text{``}string\text{''} \mid \text{``}dict\text{''} \mid \text{``}proc\text{''} \mid \text{``}page\text{''} \mid \text{``}pmap\text{''} \mid \text{``}account\text{''} \mid \text{``}channel\text{''} \mid \text{``}cpu\text{''} \mid \text{``}sponsor\text{''} \mid \text{``}ackqueue\text{''} \mid \text{``}waitqueue\text{''} \mid \text{``}origin\text{''}) \tag{2.8}$$

The optional entries in the procedure description dictionary are described by rules 2.9–2.10. They are:

1. The entry under the name `order`: its value is either one of the names `first, last,` and `any` or a non negative integer. This value specifies the *order* of the procedure in the admitted sequences of procedure calls for

[12] It is clear that this kind of interaction between the caller and the callee requires that not only the caller trusts the callee but also that the callee trusts the caller.

interacting with the service. The value `first` indicates that the procedure is the first procedure to be called when interacting with the service. Similarly, the value `last` indicates that the procedure is the procedure to be called last, i.e., to terminate the interaction with the service. The value `any` specifies that the procedure can be called at any moment. An finally, an integer value gives precisely the position of the procedure in an admitted sequence of procedure calls.

2. The entry under the name `multi`: specifies the number of times the procedure can be called to interact with the service. The value of `multi` can be either the name `once` or the name `many` respectively for a procedure that can be called only once and a procedure that can be called an arbitrary number of times.

$$Porder \stackrel{def}{\Longrightarrow} \mathbf{key}(name(\text{``}order\text{''})); \mathbf{value}(name(\text{``}any\text{''} \mid \text{``}first\text{''} \mid \text{``}last\text{''}) \mid int(ANY)) \tag{2.9}$$

$$Pmulti \stackrel{def}{\Longrightarrow} \mathbf{key}(name(\text{``}multi\text{''})); \mathbf{value}(name(\text{``}once\text{''} \mid \text{``}many\text{''})) \tag{2.10}$$

2.3.3 The Admitted Sequence of Procedure Calls

The admitted sequence of procedure calls is formally defined by rules 2.11–2.22. It defines all the sequences of procedure calls that are valid for a proper interaction with the service. The admitted sequence of procedure calls can be expressed either "partially" or "totally" (see rule 2.11).

$$Ssequence \stackrel{def}{\Longrightarrow} (PartialSeq \mid TotalSeq) \tag{2.11}$$

When it is expressed partially, the admitted sequence of procedure calls is defined under the entry name `partial` in the SAP dictionary. Its value is then a dictionary describing the sequence (rules 2.12–2.20). Entries in the *partial dictionary* express the order (precedence relation) that must be maintained between the different procedures for interacting with the service. The partial dictionary must contain at least one of the following entries:

1. The entry under the name `first` is associated to an array. The array contains the name of procedures that may be called at the first position in a sequence of admitted procedure calls (rule 2.13).

2. The entry under the name `last` is also associated to an array of procedure names. Similarly to the `first` entry, it contains names of procedures that may be called last in a sequence of admitted procedure calls, i.e., when terminating the interaction with the service (rule 2.14).

3. The entry under the name `before` defines a "precedence" relation between various procedures. Its value is an array of sub-arrays. Each sub-array contains procedure names that appear in the interface dictionary. Each entry in the sub-array specifies that the corresponding procedure must be called before the first call to the procedure corresponding to the entry that follows it in the sub-array (rules 2.15 and 2.20). For example,

$$before \sim [[a\ b\ c]]$$

means that a call to procedure `a` must occur before the first call to procedure `b` and a call to `b` must occur before the first call to `c`.

4. The entry under the name `follow` is an array of sub-arrays containing procedure names that appear in the interface dictionary and expressing precedence relations between different procedures. Each element in a sub-array specifies that in any admitted sequence of procedure calls, any call to the procedure named by the element is immediately followed in the sequence by a call to the procedure named by the following element in the sub-array (rules 2.16 and 2.20). For example,

$$follow \sim [[a\ b\ c]]$$

means that each call to procedure `a` is immediately followed by a call to procedure `b` and each call to procedure `b` is immediately followed by a call to procedure `c`.

5. The entry under the name `precede` also defines a "precedence" relation between procedures. It has the same structure as the `follow` entry; i.e., an array of array of procedure names. Similarly to the `follow` entry, an element in a sub-array specifies that each call to the named procedure is immediately preceded by a call to the procedure named by the following element in the sub-array (rules 2.17 and 2.20). For example,

$$precede \sim [[a\ b\ c]]$$

means that each call to procedure `a` is immediately preceded by a call to `b`, and any call to `b` is immediately preceded by a call to procedure `c`.[13]

6. The entry under the name `post` also defines a "precedence" relation. It has the same structure as the `follow` entry. Each element x in a sub-array indicates that every call to the procedure x is followed by a call to the procedure corresponding to the following element y in the sub-array and that the call to procedure y must appear in the sequence before a subsequent call to procedure x (rules 2.18 and 2.20). The difference between the `follow` and the `post` entries is that `follow` requires that any call to a

[13] It is worth noting that $follow \sim [[a\ b]] \not\Leftrightarrow precede \sim [[b\ a]]$. For example, the first precedence relation allows calls to procedure `b` not followed by calls to procedure `a` while the second precedence relation does not allow them.

certain procedure appears immediately after a call to another procedure, whereas the entry `post` requires that any call to a given procedure is followed some time later with a call to another procedure and that the call to the second procedure cannot be postponed until another call to the first procedure. For example,

$$post \sim [[a\ b]]$$

specifies that any call to procedure `a` is followed some time later by a call to procedure `b` and that the call to `b` cannot be postponed until another call to `a`.

7. The entry under the name `ante` defines a "precedence" relation between various procedures, that is the dual case of the `post` entry (rules 2.19 and 2.20). For example,

$$ante \sim [[a\ b]]$$

indicates that any call to procedure `a` in the admitted sequence of procedure calls must have been preceded some time ago by a call to procedure `b` and that a call to `b` must appear between any two consecutive calls to `a`.[14]

$$PartialSeq \stackrel{def}{\Longrightarrow} \textbf{key}(name(\text{“}partial\text{”})); \textbf{value}(dict : \textbf{entries}(\textbf{many}\ FirstEntry, LastEntry, BeforeEntry, FollowEntry, PrecedeEntry, PostEntry, AnteEntry)) \quad (2.12)$$

$$FirstEntry \stackrel{def}{\Longrightarrow} \textbf{key}(name(\text{“}first\text{”})); \textbf{value}(array[1+] : \textbf{elements}(ProcName)) \quad (2.13)$$

$$LastEntry \stackrel{def}{\Longrightarrow} \textbf{key}(name(\text{“}last\text{”})); \textbf{value}(array[1+] : \textbf{elements}(ProcName)) \quad (2.14)$$

$$BeforeEntry \stackrel{def}{\Longrightarrow} \textbf{key}(name(\text{“}before\text{”})); \textbf{value}(array[1+] : \textbf{elements}(ArrayDef)) \quad (2.15)$$

$$FollowEntry \stackrel{def}{\Longrightarrow} \textbf{key}(name(\text{“}follow\text{”})); \textbf{value}(array[1+]$$

[14] Again, it is worth noting that $post \sim [[a\ b]] \not\Leftrightarrow ante \sim [[b\ a]]$. For example, $ante \sim [[b\ a]]$ allows sequences of procedure calls containing consecutive calls to procedure `a` without calls to procedure `b` in between. Those sequences are not allowed by $post \sim [[a\ b]]$.

$$: \mathbf{elements}(ArrayDef)) \quad (2.16)$$

$$PrecedeEntry \overset{def}{\Longrightarrow} \mathbf{key}(name(\text{“}precede\text{”})); \mathbf{value}(array[1+] : \mathbf{elements}(ArrayDef)) \quad (2.17)$$

$$PostEntry \overset{def}{\Longrightarrow} \mathbf{key}(name(\text{“}post\text{”})); \mathbf{value}(array[1+] : \mathbf{elements}(ArrayDef)) \quad (2.18)$$

$$AnteEntry \overset{def}{\Longrightarrow} \mathbf{key}(name(\text{“}ante\text{”})); \mathbf{value}(array[1+] : \mathbf{elements}(ArrayDef)) \quad (2.19)$$

$$ArrayDef \overset{def}{\Longrightarrow} array[1+] : \mathbf{elements}(ProcName) \quad (2.20)$$

And when it is expressed totally, the admitted sequence of procedure calls is defined under the name `total` in the SAP dictionary (rules 2.21–2.22). Its value is an array of sub-arrays. Each sub-array contains at least two elements. The first element of each sub-array is either the name `once` or the name `many`; the remaining elements are procedure names that appear in the interface dictionary. The first sub-array in the the total array contains the names of procedures that can appear in the first position of an admitted sequence of procedure calls for interacting with the service; the second sub-array contains the names of procedures that can appear in the second position of an admitted sequence of procedure calls, and so on. The name `once` in the first position of a sub-array indicates that one of the procedures corresponding to the remaining elements of the sub-array has to be called exactly once; and the name `many` specifies that a number of procedures corresponding to the remaining elements of the sub-array can be called an arbitrary number of times.[15]

$$TotalSeq \overset{def}{\Longrightarrow} \mathbf{key}(name(\text{“}total\text{”})); \mathbf{value}(array[1+] : \mathbf{elements}(OrderArray)) \quad (2.21)$$

$$OrderArray \overset{def}{\Longrightarrow} array[2+] : \mathbf{elements}(\mathbf{first}(name(\text{“}once\text{”} \mid \text{“}many\text{”})), \mathbf{other}(name(ProcName)) \quad (2.22)$$

[15] We can consider a sequence of procedure calls as a sequence of patterns where procedure names are the patterns. Then there is a similarity between regular expressions and admitted sequences of procedure calls expressed totally. The keyword `many` in a sub-array of the `total` entry has an effect similar to that of the * symbol a regular expression.

Examples

We present in this section two examples to illustrate how the admitted sequence of procedure calls may be expressed for a service.

As a first example, we consider again the distributed semaphore service already mentionned in this dissertation. This service provides an interface consisting of three procedures: the `create` procedure that creates a new semaphore, the `P` procedure to acquire a semaphore and the `V` procedure to release a semaphore. The admitted sequence of procedure calls for this service may be expressed partially by the following relation (entries in the partial dictionary):

$$first \quad \sim \quad [create]$$

The only precedence constraint is that a call to `create` must appear first in the sequence of procedure calls. There are no constraints imposed on the `P` and `V` procedures other than the fact that they cannot be the first called procedures. A more complete interface would therefore use the optional entry in the interface dictionary to specify that the `multi` entry for the `P` and `V` procedures has the value `many`; i.e., that an arbitrary number of calls to `P` and `V` appear in any order after the call to `create`. The value of `multi` for the `create` procedure is `once`.

When expressed totally, the admitted sequence of procedure calls for the same service is:[16]

$$total \quad \sim \quad [[once\ create]\ [many\ P\ V]]$$

Our second example is the distributed shared memory service that is discussed in detail in chapter ??. This service provides three procedures to interact with it: the `dsmcreate`, the `dsmput` and the `dsmget` procedures. The admitted sequence of procedure calls for this service may be expressed partially by the following relations:

$$\begin{aligned} first &\quad \sim \quad [dsmcreate] \\ before &\quad \sim \quad [[dsmput\ dsmget]] \end{aligned}$$

This imposes two constraints: (a) a call to the `dsmcreate` procedure must be issued first and (b) before the first call to the `dsmget` procedure, a call to the `dsmput` procedure must be issued. As for the first example, the value of the `multi` entry in the interface dictionary is `many` for both the `dsmput` and `dsmget` procedures, while this value is `once` for the `dsmcreate` procedure.[17]

[16] In the regular expression notation where procedure names correspond to patterns, this is equivalent to $create\ (P \mid V)^*$.

[17] A more strict and more meaningful expression of the admitted sequence of procedure calls for the distributed shared memory service would be:

$$\begin{aligned} first &\quad \sim \quad [dsmcreate] \\ follow &\quad \sim \quad [[dsmput\ dsmget]] \end{aligned}$$

This requires that a call to `dsmput` be followed immediately by a call to `dsmget`; i.e., it is meaningless to write to memory a value that is not be read afterwards.

And when expressed totally, the admitted sequence of procedure calls for the distributed shared memory could be:[18]

$$total \quad \sim \quad [[once\ create]\ [once\ dsmput]\ [many\ dsmput\ dsmget]]$$

First, the `create` procedure is called, then the `dsmput` procedure is called to initialize a shared data item, and finally, the `dsmput` and the `dsmget` procedures can be called in any order and any number of times.

2.3.4 Discussion

Our approach for the implementation of distributed services requires that each service describes precisely its service access point. This imposes an additional constraint on the service client in the sense that it must interpret correctly the information found in the service's SAP in order to interact appropriately with the service. In this respect, the task of interacting with a service becomes more complex from the client's point of view as compared to an approach where the service interface is known at development time. The advantage gained from this approach is that it is no longer necessary to know the interface of a service at development time in order to use the service correctly. Our approach can therefore be seen as allowing to "late bind" an interface to a service. Furthermore, it allows to change dynamically the interface of a running service.

So far, we can consider that the above language describes the *operational interface* of a distributed service. It allows to define the information required by the service user to use the service correctly. A wide open environment can accomodate a large number of distributed services. In such an environment, one can expect a number of services to vary only in subtle ways. Moreover, different implementations of a service can coexist. It becomes essential (crucial) in such an environment for a user (distributed application) to choose the appropriate implementation of the right service to accomplish efficiently and in a cost effective way a given task. To allow the users choose the appropriate services they need, the messenger language should be extended to describe fully also the semantics of a service; i.e., to define precisely the `semantical interface` of a service.

Finally, a distributed service must have also a *management interface*. This interface is intended for the interaction between a service and its manager (for example a software manager). The messenger interface language should be extend to support a simple way for describing the management interface of a distributed service.

2.4 Dealing with Up/Down Dynamicity

The state of an application—its data and flows of control—in a messenger environment is expected to be distributed in different messenger execution envi-

[18] In the regular expression notation where procedure names correspond to patterns, this is equivalent to $create\ dsmput\ (dsmput \mid dsmget)^*$.

ronments running themselves in different hosts. Endeed messengers will room through the network searching for the appropriate resources and services they need to complete their task. More over, one cannot expect a large environment to be static. On the contrary, the number of platforms and hosts will vary dynamically as a consequence of some hosts coming up and others going down. However, despite this dynamicity, one should expect an application to behave in a consistent way; i.e., without corrupting or loosing part of its state. Therefore, implementations of distributed services should be enough robust to take into account the dynamic nature of the environment.

By up/down dynamicity, we mean (a) the ability for new messenger platforms to join a system; i.e., to bring additional resources to messengers running in the system and (b) the ability to remove some messenger platforms from a running system, and therefore removing some resources from the system. As an example, in a large environment, some hosts can be shut down for maintenance or for software upgrade. The remaining part of the system should continue to run without loosing or corrupting the state of the system. Here after, the *up event* is used to designate the event of a messenger platform coming up and the *down event* is used for the event of a messenger platform going down.

Beside the dynamicity induced by messenger execution environments coming up or going down, services themselves can also be dynamic. Services will be made available or will disappear from a platform as the result of the execution of messengers in that host. One typical situation is when a messenger is no longer able to pay for the resources it consumes for its execution in a given platform. Such a messenger will disappear from the platform with the different services it offers. It will probably move to another platform where resources are cheaper or it will completely disappear from the system.

Messengers seem well suited for the implementation of fault tolerant distributed applications. Indeed, messengers allow to easily distribute the state of an application on different hosts. To achieve fault tolerance, however, one needs appropriate protocols to handle different crash and failure conditions. These protocols are often costy and have to be combined with other costy techniques to handle effectively fault tolerance. Handling up/down dynamicity is not handling neither failure nor crash conditions. On the contrary, up/down dynamicity is intented to handle situations where platforms do not leave the system suddenly. In fact up/down dynamicity is restricted to situations where each platform leaving the system is given a "grace period" to allow it to execute some crucial tasks in order to maintain the system in a coherent state. This is the case for example when hosts are shut down for maintance or software upgrade purposes.

2.4.1 Handling Up/down Dynamicity

As large environments are expected to be highly dynamic, service providers should provide up/down dynamic implementations of their services. We propose in this section, enhancements to our service architecture that allow services and applications to deal with this dynamicity.

Our approach for dealing with up/down dynamicity is consistent with the choice to have messenger execution environments provide only local services to messengers. Therefore, messenger platforms will provide only basic mechanisms to allow messengers themselves manage system dynamicity. In this way, there are no hardwired protocols inside the messenger execution platforms. This ensures a high degree of flexibility. Here again, we rely on a number of conventions to be used on one part by all the messenger platforms (the H conventions) and on the other part, by all the messengers (the A conventions).

Convention M 2.3 *A service provider wishing to be informed when platforms come up or go down should provide as part of its service interface two procedures named respectively* `up` *and* `down`*. The* `up` *procedure is intended to be executed by a platform coming up as part of its bootstrap process. The* `down` *procedure is intended to be executed by a host going down as part of its shut down process. The* `up` *and* `down` *procedures can be executed concurrently by different platforms.*

Convention P 2.2 *When a platform comes up, it should as part of its bootstrap process, locate all the services available in the system. It should then execute the* `up` *procedure of each service providing that procedure. When a host goes down, it should as part of its shut down process locate all available local services. It should then execute the* `down` *procedure of each of those services providing such a procedure.*

The mechanism offered by messenger platforms to allow service providers and application developers manage system dynamicity is the possibility to install the `up` and `down` procedures which behave as triggers. These triggers are activated at platform bootstrap for the `up` procedure or at platform shut down for the `down` procedure. In this way, service providers are informed of the asynchronous up and down events of different platforms without the need to poll for those events.

Service providers decide themselves if they ought use the up/down mechanism offered by the platforms. In this case, they are responsible for providing the procedures implementing the appropriate protocols which actually manage and handle system dynamicity. These procedures should be responsible of:

- locating new available platforms and resources they bring into the system;
- distributing the service or application state (data and flows of control) on different platforms according to the resources available in each platform;
- executing crucial actions and migrating data and flows of control according to a prestablished policy in order to maintain the coherence of the service or application;
- ensuring that up/down events are actually due to platforms coming up or going down and not to some messengers trying willingfully or not, to break down the service or its security.

Handling up/down dynamicity can require complex cooperation protocols between different hosts. Due to their complexity, it can be difficult to add these protocols to a running service. Adding dynamicity to a running service will certainly require to review the overall design of the service, most of the time. Hence, services should be designed with up/down dynamicity in mind rather than trying to add dynamicity to them after hand.

Designing effective and efficient up/down dynamic services can be a challenging task. Here are some aspects the service designer should consider when designing and implementing up/down dynamic services:

- Up/down events are expected to occur rarely for a given platform. And when a platform comes up, it is expected to run for a long time before going down. Up/down events should therefore be treated as marginal situations for a running service and should not degrade significantly the performance of the service. The execution overhead introduced for the management of up/down events should remain low.

- Although up/down events are expected to occur only rarely, the correct behavior of a service in a dynamic environment will depend on an effective handling of those events. This suggests that the service designer should be careful to handle all conditions which can arise, even the most improbable ones. The service designer should be particularly careful to handle correctly the following cases: (a) an up event followed "immediately" by a down event from the same messenger platform, (b) messenger platforms coming up concurrently and (c) messenger platforms coming up while others are going down.

- The invocation of a service can fail due to the hosting platform going down. This event should be signalled to the service user as soon as possible to allow the service user move to another platform where it can retry to reinvoke the service.

- A messenger platform that has gone down can come up again on a latter time and rejoin the system. One can not expect that when the platform comes up again, it will contain the same resources and offer the same services. Indeed, resources and services available in a messenger platform will vary from platform to platform according to the messengers that has visited the platform.

2.4.2 Security Issues

When the up/down mechanism is used to manage system dynamicity, the designer of a service should be aware of the potential security problems relative to the `up` and `down` procedures. As stated above, the `up` and `down` procedures are intented to be executed respectively only when up and down events occur. However ill-intentioned messengers can call the `up` and `down` procedures to signal "false" up and down events to the service. False up and down events can break down a service or at least disturb its normal behavior.

On one hand, a high rate of false up events will considerably degrade the effeciency of a service as the service will spend a large part of its processing time handling those events. It is clear that a sufficient high rate of false up events can make a service useless especially if interactions initiated by users of the service are delayed. To limit somewhow the harmful effect of false up events, an implementation should not delay interactions from users in order to handle up events. On the contrary, interactions from users should be given priority over up events. It is even preferable to interrupt or to abort the handling of an up event when an interaction from a user occurs, and to switch back again to the handling of the up event when no other user interactions are pending.

On the other hand, false down events will surely break down a service. Indeed, when a down event occurs, the state of the service is removed from the platform going down and moved to another platform. In other words, the service is "shut down" as a consequence of a down event. Therefore, a number of false down events will result in the service being completely removed from all the messenger platforms—the service disappears from the system.

Hence a service implementation must distinguish "true" up and down events from "false" events generated by ill-intentioned messengers, only in the aim to be harmful. Above all, an up/down implementation of a service must be particularly careful to identify precisely false down events as they are potentially more harmful for the service.

On way a service can protect itself is to restrict the execution of the `up` and `down` procedures to authorized messengers only. This can be done by having these procedures request the caller to provide a secret agreed upon before hand. The protection scheme presented below is based on the following convention:

Convention P 2.3 *Each messenger platform should maintain a pair of keys—a private key and a public key—used respectively to decode and to encode data intended to be "understood" only by the platform and the data provider. The public key should be accessible in a uniform way (for example under a common name) to all messengers running in the platform.*

The main idea here is to restrict the execution of the `down` procedure only to a messenger which has successfully executed the `up` procedure previously. When the `up` procedure is executed, the service generates a secret key and encodes it using the public key of the platform executing the procedure. The encoded key is passed to the platform which will use it later when it calls the `down` procedure. And when the platform goes down, it decodes the key received upon the execution of the `up` procedure and passes it as a parameter to the `down` procedure. As the secret key required by the `down` procedure was previously encoded by the public key of the platform, one can expect this key to be decoded only by the private key of the same platform.

For the above approach to work, the service must obtain somehow the public key of the messenger platform coming up and executing the `down` procedure. This will be the case if all messenger platforms are registered and certified by some trusted authority which also maintains a database of messenger platform

public keys. Moreover, we assume that messenger platforms that are registered and certified by the above authority are also trusted and that they cannot generate false up and down events.

The above approach can work only for those platforms that come up after the service is installed in the system, i.e., after the service is running. When installing the service on an already running messenger platform, the `up` procedure cannot be used to pass to the messenger platform the secret key required by the `down` procedure. In this case, a simple variation of the above approach can be used. It is based on the following convention:

Convention M 2.4 *A service implementing the* `down` *procedure should limit the execution of that procedure to the messenger platform. If the service is installed in an already running host, it should store in the service access point under the name* `downkey`*, a secret key encoded by the platform public key. The corresponding decoded key should be provided as a parameter to the* `down` *procedure.*

In the following two subsections, we summarize the desired functionality that should be provided respectively by the `up` and the `down` procedures.

2.4.3 The up Procedure

The main aim of the `up` procedure is to discover new hosts joining the system and to make the service available in these hosts for the messengers that will visit them. Moreover, the `up` procedure can be used to distribute the state—data and control flow—of the service on different platforms trying to optimize globally the use of available resources in the system (increase the performance and the efficiency of the service).

It is therefore preferable to have the `up` procedure completely decoupled of the protocol that ensures the coherence of the service. This means that, if for some reasons, the `up` procedure is not executed when an up event occurs, the service should continue to run correctly. In fact, if the `up` procedure is completely decoupled from the coherence protocol of the service, a service can implement only partly up/down dynamicity by providing only the `down` procedure. In this case, the service can completely ignore messenger platforms coming up after the service has been installed or an alternative mechanism can be used to allow the service spread in the entire network of messenger platforms.

Having the `up` procedure completely decoupled from the service's coherence protocol allows the service to give user interactions priority over up events as discussed above. Up events are handled only when no user interactions are pending.

The `up` procedure should provide at least the following functionality:

- verify if the procedure has been called by a new platform coming up, i.e., that there is actually an up event and if so, obtain the address, the public key and the identity of the new messenger platform;[19]

[19] The up procedure should abort executing if it is not provided the right secret key.

- generate a secret key, encode it using the public of the platform coming up and pass the encoded key to the platform;
- make the service available in the new platform by registering it and installing it so that it can be used by messengers that will reach the platform;[20]
- install in the new platform a `down` procedure in the service's SAP so that the service can be notified when the platform is shut down;
- additionally, move part of the service state to the new platform to benefit from the resources it contains in order to optimize service efficiency.

2.4.4 The `down` Procedure

The `down` procedure is undoubtedly the most important, but also the most complex of the two procedures for managing system dynamicity. This is because, contrary to the `up` procedure, the `down` procedure cannot be decoupled from the protocol ensuring the coherence of the service. In fact the aim of this procedure is to ensure that a service remains coherent after a messenger platform has been shut down. Therefore, every service using the up/down mechanism to manage system dynamicity should provide at least a `down` procedure.

Down events should be considered by the service as high level priority events which should be handled before all other events. Failing to handle a down event can result in the loss of part of the service state. For this reason, a service implementation can allow the handling of down events to interrupt all other events, even interactions with the users of the service.

The `down` procedure should provide at least the following functionality:

- verify if the down event really corresponds to a messenger platform going down;[21]
- migrate the local state of the service to other messenger platforms according to the policy enforced by the service;

[20] There are two ways of installing an up/down service in a messenger platform: The first alternative is to have the `up` procedure registered and installed in only one host among the hosts where the service is installed. When a new messenger platform comes up, the up event must reach the unique host where the `up` procedure is installed in order to execute the `up` procedure. This seems simple, however, additional work has to be done by the `down` procedure when the platform where the `up` procedure is installed goes down—migrating the `up` procedure to another host, handling concurrent down and up events when the down event is generated by the host containing the `up` procedure becomes more complex.

The second alternative is to have the `up` procedure registered in every host where the service is available. In this latter case, multiple copies of the `down` procedure are executed on each up event. Therefore, the different instances of the `up` procedure should cooperate to do their job correctly.

[21] The `down` procedure insures that it is called by a trusted messenger platform by examing the validity of the secret key provided by the caller. Execution should be aborted if the secret key is not valid.

- signal other messengers providing the service in other platforms that the service has been shut down.

2.4.5 Summary

Large distributed environments are expected to be dynamic and services should adapt themselves to dynamic environments. Up/down dynamicity refers to the dynamicity induced by (a) messenger platforms coming up—up events—and joining a running system and (b) messenger platforms leaving the system—down events.

Messenger platforms provide a simple mechanism for signalling up and down events to registered services. Services can use this simple mechanism to manage system dynamicity. A service registers a `up` procedure that is executed whenever a messenger platform comes up and a `down` procedure that is executed whenever a messenger platform goes down. These procedures must be protected against false up and down events and must implement the policy enforced by the service to deal with system dynamicity.

Messengers which do not implement services can still use the up/down mechanism provided by messenger platform to be notified of up and down events. They can publish dummy services (must choose appropriate names for them, e.g., randomly generated by the system) and attach to the service a down and/or an procedure.

Bibliography

[BLC95] T. Berners-Lee and D. Connolly. *Hypertext Markup Language—2.0*, November 1995. RFC no 1866.

[BLFN96] T. Berners-Lee, R. Fielding, and H. Nielsen. *Hypertext Transfer Protocol—HTTP/1.0*, May 1996. RFC no 1945.

[Blo92] John Bloomer. *Power Programming with RPC*. UNIX Networking Programming. O'Reilly & Associates, Inc., 1992.

[DMMTH95] Giovanna Di Marzo, Murhimanya Muhugusa, Christian F. Tschudin, and Jürgen Harms. The Messenger Paradigm and its Impact on Distributed Systems. In *ICC'95 workshop on Intelligent Computer Communication*, 1995.

[Nel81] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Carnegie-Mellon University, 1981.

[PR85] J. Postel and J. Reynolds. *File Transfer Protocol*, October 1985. RFC no 959.

[Tsc93] Christian F. Tschudin. *On the Structuring of Computer Communications*. PhD thesis, Université de Genève, 1993. Thèse No 2632.

[Tsc94] Christian F. Tschudin. An Introduction to the M0 Messenger Language. Technical Report No 86 (Cahier du CUI), University of Geneva, 1994.