DISTRIBUTED SEMAPHORE IN A MESSENGER ENVIRONMENT[*]

Murhimanya Muhugusa, Giovanna Di Marzo, Christian Tschudin, Jürgen Harms
University of Geneva, e-mail: muhugusa@cui.unige.ch
URL: http://tioswww.unige.ch/tios/msgr/home.html

ABSTRACT

The messenger paradigm advocates the exchange of programs called messengers between communicating hosts instead of messages. Each host contains a messenger execution environment called messenger platform. Messengers are expressed in a language understood by all the platforms. A distributed messenger environment is a collection of messenger platforms linked through an unreliable network offering only a datagram service. Provision of coordination services in such an environment has to be handled in a new way since threads of controls are mobile (messengers move from host to host). The classical client/server paradigm based on a data exchange mechanism is not suitable for a messenger environment. In this paper we discuss how communication services might be provided in messenger environments using as an example the distributed semaphore service in the MØ *distributed messenger environment. A distributed semaphore allows messengers executing on different hosts to synchronize their execution independently of their physical location.*

Keywords: distributed semaphore, messengers, distributed mutual exclusion, MØ.

1 INTRODUCTION

Distributed computing is gaining more and more importance. Different environments for the implementation and the execution of distributed applications such as CORBA [2] and DCE [9] have been built on top of existing operating systems. Modern operating systems and kernels equally (CHORUS [10], MACH [1], AMOEBA [11]) offer some features for the development of distributed applications. And recently, mainly in the arena of distributed AI, software agents are used as the basis for distributed applications. Most of the above attempts to master distributed computing are based on a message exchange mechanism and the well known RPC paradigm [8]. Distributed software is structured in clients and servers which exchange messages which are interpreted using a pre-established protocol.

The messenger paradigm introduced by Tschudin [12] in the arena of computer communications allows the communication between hosts without the need of a pre-established protocol. The host initiating the communication will send to its partner the data and the necessary rules (code) to interpret it. This is an instructional approach as compared to the interpretative approach of messages. We are studying the impact of this paradigm on distributed applications [5] and we are using it in the development of a distributed operating system [14]

While the RPC paradigm and the client/server programming model for distributed computing based on it are suitable for provision of services when a message passing mechanism is used for computer communication, they are not suited in environments where computer communication is based on the exchange of programs. This paper presents an approach for service provision in a messenger environment. However, techniques developed in this paper are also relevant to other higher-level mobile software agents which may need inter-node synchronization.

The distributed semaphore service is used in this paper to illustrate the techniques developed for service provision. The semaphore paradigm has been introduced by Dijkstra [3] to synchronize the execution of concurrent processes on a mono-processor system. A semaphore is used to grant exclusive access to a “critical region” of a program (usually where shared data has to be accessed without racing). The semaphore paradigm has been extended for usage on multi-processor systems [4, 6], usually those with shared memory, to guarantee a consistent access to shared data. While on this kind of machines the semaphore is considered to be a low-level service which is provided at the kernel-level using special processor instructions ensuring atomicity (“atomic test and set” or “spin lock”), the distributed semaphore

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

service described in this paper is a high-level (inter-node) service implementation.

Section 2 presents service provision in a messenger environment. Section 3 introduces the messengers used to achieve the distributed semaphore service. Section 4 is devoted to the discussion of the distributed semaphore service and we give a brief summary in section 5.

2 Provision of Services in a Messenger Environment

In this section we present the basics of messenger environments, the role of common conventions between messenger platforms for the provision of services and an architecture for the provision of services in such environments.

2.1 A Messenger Environment

In a distributed messenger environment, hosts communicate by exchanging messengers only, i.e., mobile code fragments. Each host implements a messenger platform responsible for the execution of messengers. The different platforms are linked through unreliable communication channels and form together the distributed messenger system. The main characteristics of such a system are summarized below:

- A messenger having reached in full and correctly a platform becomes an independent concurrent thread of execution, i.e., no other thread can stop it, or kill it. All the platforms share a common language for expressing messenger behavior and a common external representation for the physical exchange of messengers;
- Messengers executing in a platform can share common data through a global store and can synchronize their execution through process queues;
- All the platforms use common conventions for locating and accessing local resources and services.

We have implemented two different messenger environments: MSGR-S [15] based on SCHEME and `MØ` [13] which has inherited from POSTSCRIPT [7] its stack orientation, its postfix notation, its syntax and some of its operators and data structures. C-like pseudo-code together with `MØ` operators and data structures presented below are used in this paper to describe our algorithms.

In `MØ`, the global store is represented as a couple of “dictionaries”: the `globdict` and the `servdict` dictionaries. Other dictionaries can be created by the `dict` operator. Data is stored in a dictionary by the `define` operator and accessed by the `get` operator, in the two cases, using a key. The `globdict` is not browsable; i.e, one must know the exact key associated to a data to access it, but any process can read, write and modify the dictionary, while the `servdict` is browsable, i.e., one can moreover construct the list of all keys and their associated values. Keys can have an arbitrary data type but most of the time they will be arbitrary bit streams created with the `random` operator. The `knowndict` operator allows to check if a given data exists in a dictionary. Data defined in a dictionary by the `define` operator can be removed from the dictionary with the `undef` operator by any messenger having access to it (knowing the key used to define it). For this reason, `MØ` provides the `secretdef` and `secretundef` operators for securing data. The former adds data in a dictionary by associating with it two keys: a secret-key and a public key. The public key is used only for accessing the data while the secret-key is mandatory for removal of the data from the dictionary. The removal of the data is done by the `secretundef` operator. A messenger can then publish data for access to every messenger or a family of messengers and still restrict its removal from the dictionary to only those messengers to which it has handed the data secret-key.

In order to coordinate the access to the global store, messengers use process queues. Four operators are used to handle process queues: the `enterqueue` operator allows a messenger to insert itself in a queue, the `stopqueue` operator is used to freeze a queue and therefore to block all messengers inside it, the `startqueue` is used to unblock a queue previously stopped while the `leave` operator removes the calling messenger from its queue. A messenger can be in at most one queue at a time and once in a queue, a messenger will be blocked until it reaches the head of the queue. The `enterqueue` operator can be used with a timeout value which expresses the maximum amount of time the messenger may remain blocked in the queue. Finally we mention the `submit` operator that is used to create a new messenger for execution either in the local or in a remote host.

2.2 The Role of Conventions

In our distributed messenger environment, each platform offers only local services. Inter-node services are implemented with messengers by letting messengers move through the network looking for the appropriate resources, data or information to achieve the desired service. Hence, a messenger must be able to locate resources, data and services and use them appropriately. For this, conventions are necessary: (a) common conventions shared by all the hosts for locating and accessing the resources available in a platform and (b) uniform conventions between service providers and service consumers for publishing/discovering services and using them appropriately. More conventions can be used in a network of platforms to achieve the desired effect.

We propose in the next section, a service architecture based on three sets of conventions: the two sets of conventions cited above (for locating resources in a host, for publishing/discovering and using services) and a set of conventions for managing dynamicity in a distributed system. By dynamicity we mean the adjunct of hosts to a system (service community) or the removal of hosts from it.

2.3 A SERVICE ARCHITECTURE FOR MESSENGERS

The service architecture we develop here is based on the assumption that a distributed system is dynamic, i.e., new hosts can come up, attach themselves to the running system, while some hosts can go down. The architecture does not support the case where hosts go suddenly down due, for example, to a failure or crash. We restrict ourselves to the simpler case, where each host leaving the system is given a grace period during which it can execute the important tasks to maintain the system in a coherent state. This is the case for example when hosts have to be shut down for maintenance or for software upgrade.

LOCATING RESOURCES, PUBLISHING AND DISCOVERING SERVICES

The `servdict` plays an important role in publishing and locating services in a `MØ` environment. This is a public browsable dictionary where messengers can publish services they offer to other messengers. A messenger publishing a service in this dictionary chooses a public service name, usually a string, which will be used by the service consumers to locate the service. Since `servdict` is a public dictionary and thus accessible to all messengers, service providers must publish their services in a secure way; i.e, without having other messengers remove their entries. For that, firstly, the service provider adds the service name in `servdict` with the `secretdef` operator. This operator takes as arguments a secret-key, let us say `k1`, and the service name. It creates a public-key `k2` using a function

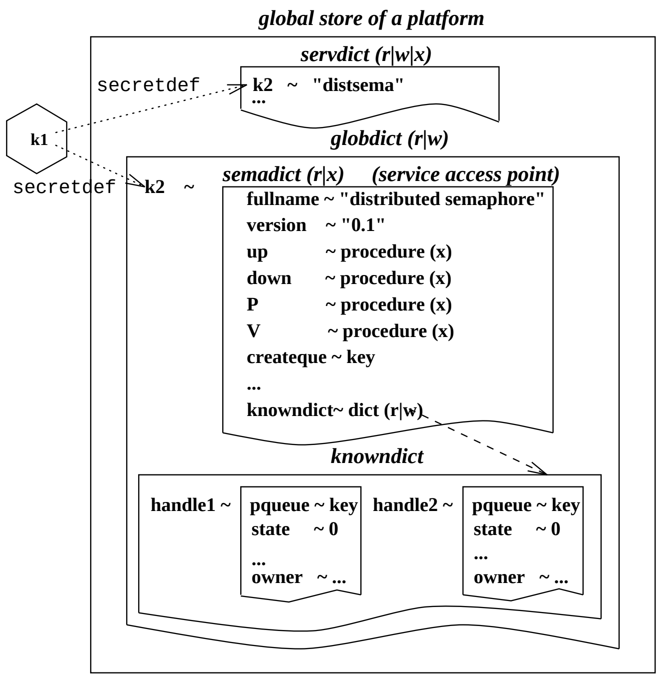

Figure 1: Publishing and locating a service

based on DES and adds in `servdict` the association between `k2` and the service name. The association (`k2`, service name) is accessible to all messengers but can be destroyed only by the `secretundef` operator if the secret-key `k1` is provided as an argument. Thus by keeping the `k1` key secret, no other messenger can wipe out the trace of the service being defined. Next, the service provider creates a private dictionary with read-only access to other messengers, and adds it in `globdict` using the `secretdef` operator with the same secret-key `k1`. As above, this results in the association (`k2`, private dictionary) in `globaldict`. Now, the same key `k2` appears both in `servdict` where it is associated with the service name and in `globaldict` where it is associated with the private dictionary. The service provider will store in the private dictionary all the necessary information to interface with the service; i.e, the private dictionary becomes what we can call the "service access point" for the service being defined.

Figure 1 shows how the distributed semaphore can be published. In step one, the name `distsema` is chosen to identify the distributed semaphore service and is added in `servdict` by `secretdef` provided with the secret-key `k1`. In step two, the `semadict` dictionary is created, initialized with the appropriate information to interface with the service being published and `secretdef`ed in `globdict` using the same key `k1`.

A service consumer accesses the service through the service access point. The question is thus how to locate the service access point of a known service. The consumer uses the service name to browse the `servdict` dictionary and locates the association (`k2`, service name). It then uses the `k2` key to locate and access in `globaldict` the service access point. As `globaldict` is not a browsable dictionary, the consumer must first find `k2` before accessing the service access point.

This simple convention for the publication of services allows (a) messengers to publish their services in a secure way, i.e., without having other messengers wipe out any trace of the services; and by providing simple primitives in `MØ` messengers can make portion of their code accessible in a controlled way to other messengers and (b) messengers reaching a given platform can discover the available services and interface with them appropriately.

Dynamicity

Our approach to support dynamicity avoids "polling". The running hosts do not check periodically the system for new hosts coming up or for hosts going down. Instead, the "attachement convention" is used. Under this convention, a host coming up announces its interest to attach to the system, i.e., to attach to some of its services. The other hosts will take the appropriate actions to extend available services to the new host in a consistent way. And when a host has to be shut down, it also announces this fact to the system which will "detach" the host from the system by maintaining system consistency.

Indeed, a service provider can request to be signaled whenever a new host wishing to use the service comes up. For that, the service provider installs in the service access point a procedure called `up`. When a new host comes up it submits "discovery" messengers to other hosts. These messengers will locate and attach to services (i.e, use appropriately) available on remote hosts. For each service the discovery messenger will execute its associated `up` procedure. The execution of the `up` procedure will result for example, on a "signal" being sent to the service provider. It can therefore take the appropriate actions to extend its service to the new host, for example by submitting "initialization" messengers which will populate the new host and initialize the service. Similarly, a service provider will install a `down` procedure in the service access point to be executed by hosts going down. The host going down sends "detach" messengers to remote hosts.

This simple mechanism can be used to maintain the network topology of the system and therefore to adapt the routing function and information to the network topology. The complexity for connectivity (attaching and detaching), migration and routing has to be handled by the `up/down` procedures and the messengers they create for this purpose.

3 The Distributed Semaphore Messengers

The distributed semaphore service is an inter-node service which allows messengers to synchronize their execution independently of their physical location. It is a distributed service because different messengers running on different platforms coordinate their work to achieve the semaphore. Actually our distributed semaphore service is a "coherence protocol" which mimics the functionality of a semaphore in a shared memory multi-processor system.

3.1 Basic Assumptions

When messengers want to synchronize their execution or to protect common data against racing, the messengers create a semaphore and use the semaphore service to manage it. Before entering the critical region of the program where synchronization is needed or before accessing the data, a messenger "acquires" the semaphore by executing the `P` procedure, ensuring exclusive access to the region or data. And after using the data or leaving the critical region, the messenger must "release" the semaphore with the `V` procedure so that other messengers competing for it can get a chance to acquire it. A semaphore can be used by messengers executing on different hosts; each host maintains its own local copy of the semaphore and the `P` and `V` procedures ensure that coherence is maintained between all the copies of the semaphore.

3.2 An Overview of the Protocol and the Messengers used

Each semaphore is managed as a binary token and is uniquely identified in the system by a semaphore handle returned on the host which created the semaphore. The messenger which requested the semaphore creation receives the semaphore handle and is responsible for advertising it to the other messengers needing it. At any time, a semaphore is owned by one host (platform); and only messengers executing on that platform can acquire directly the semaphore. If a messenger tries to acquire a semaphore owned by a remote platform, it first sends to the semaphore owner a messenger to request the semaphore ownership; the semaphore ownership changes therefore from host to host according to the messengers requests to acquire it.

The semaphore owner is found by maintaining on each host the value of the probable owner of the semaphore. The “true owner” of the semaphore is found by following the sequence of probable owners; i.e, a messenger is sent to the semaphore probable owner and if this host is not the true owner of the semaphore, the messenger moves itself to the probable owner and so on, until the messenger will reach the actual semaphore owner. The discovery of a semaphore owner and the acquisition of the semaphore ownership constitute the heart of the coherence protocol.

We will use the notation $(h_i \, \mathcal{R}^s_{\text{own}} \, h_j)$ to express the fact that on host h_i the probable owner of semaphore s is host h_j. If $(h_i \, \mathcal{R}^s_{\text{own}} \, h_i)$ holds, then host h_i is the true owner of semaphore s.

3.2.1 Data Structures

Each platform using a semaphore maintains local data for the semaphore on which are based all decisions taken by messengers executing on the host. The problem thus, is that of maintaining consistency between all the copies of the semaphore residing on the different hosts.

A semaphore is represented by the following information:

1. The state of the semaphore (FREE, LOCKED or REMOTE). When a semaphore is FREE or LOCKED, it is ”owned” by the local host. Otherwise it is owned by a remote host. A LOCKED semaphore is being used by a messenger while a FREE semaphore is ready for usage by any messenger having a reference to it;
2. The probable owner of the semaphore; this is the local host when the state of the semaphore is not REMOTE;
3. The `pqueue` queue used to serialize the actions of the messengers trying to acquire the semaphore;
4. The `accessqueue` and `acqhostqueue` queues for synchronizing access to the local data for the semaphore to avoid racing;
5. The identity of the (`acqhost`) host trying to acquire the semaphore ownership.

When a messenger requests the creation of a semaphore, local data is created for the semaphore, but remote hosts are not informed of this fact. Upon a successful creation of a semaphore, the semaphore creator (host on which the request is executed) becomes the first semaphore owner and returns a `handle` which allows all the hosts to determine the semaphore creator. A remote host will know a semaphore if a messenger carries there a reference to the semaphore (handle) and then the same or another messenger requests that an `P` or `V` operation be done there on the semaphore. It is only at that time that the remote host will create its local data for the semaphore; and the first probable owner of the semaphore on the remote host will be the semaphore creator.

Hereafter, we present in detail the `P` and `V` procedures for interfacing with the distributed semaphore service.

3.3 The P Procedure in Detail

The `P` procedure is called to acquire a semaphore. First the calling messenger inserts itself in the `pqueue` queue. This ensures that only one messenger can proceed on a host in its attempt to acquire the associated semaphore. Two cases can arise:

1. If the semaphore is `FREE`, the messenger acquires it, changes the semaphore state to `LOCKED` and stops the `pqueue` queue so that all other messengers trying to acquire the semaphore on the same host remain blocked in the queue. These messengers will eventually be unblocked when a `V` operation will be done on the semaphore.
2. If the semaphore is in the `REMOTE` state, the messenger stops the `pqueue` and `submit`s a new messenger (see the `msgrP`) to the semaphore owner to acquire the semaphore ownership (see the `remoteP` procedure) and waits in a queue to be unblocked when the semaphore ownership has been acquired. Then the calling messenger proceeds as in the first case. All the work of locating the semaphore owner and acquiring the semaphore ownership is done by the `msgrP` messenger. This messenger is actually the heart of the protocol and is described below. First, the code for the service access procedure `P` is given.

```
P(key, handle)
{
   serv = searchservice("distsema")
   if (serv == NULL){
      define(globdict, key, ERROR)
   }
   else {
      enterqueue(serv.createqueue)
      semadict = serv.dict
```

```
    if (!known(handle, semadict){
      # create local data for semaphore
      sema = dict()
      sema.pqueue = random()
      sema.owner = gethost(handle)
      sema.accessqueue = random()
      sema.state = REMOTE
      sema.acqhostqueue = random()
      sema.acqhost = NULL
    }
    leave()
    enterqueue(sema.pqueue)
    stopqueue(sema.pqueue)
    enterqueue(sema.accessqueue)
    if (sema.owner == THISHOST){
      sema.state = LOCKED
      leave()
      define(globdict,key,OK)
    }
    else {
      leave()
      key1 = random()
      remoteP(sema, handle, key1)
      res = get(globdict, key1)
      if (res == OK) {
        enterqueue(sema.accessqueue)
        sema.state = LOCKED
        sema.owner = THISHOST
        leave()
      }
      define(globdict, key, res)
    }
  }
}
```

The `remoteP` procedure is straightforward: the calling messenger `submit`s a `msgrP` messenger to the semaphore owner and waits for the result by inserting itself in a stopped queue which has to be restarted by the acknowledge `ackP` messenger. If after a timeout period no result is obtained, a retransmission of the `msgrP` messenger occurs.

```
remoteP(sema, handle, key)
{
  myqueue = random()
  ori = random()
  define(globdict, ori, THISHOST)
  define(globdict, key, HOSTID)
  qack = random()
  stopqueue(myqueue)
  enterqueue(sema.accessqueue)
  dest = sema.owner
  leave()
  while(1) {
    submit(dest, msgrP(ori, qack, myqueue,
      key, handle, THISHOST))
    enterqueue(myqueue, timeout)
    if (!timeout)
      break
  }
}
```

The `msgrP` messenger sent by the `remoteP` procedure can be lost; this is also true for the `ackP` messenger sent by a `msgrP` messenger to its origin. For this reason, the `remoteP` procedure keeps sending `msgrP` messengers until an `ackP` messenger is received by the local host. Thus multiple copies of the `msgrP` and `ackP` messengers can reach a host. The different copies of the same messenger must interact to coordinate their work.

All copies of the same `msgrP` messenger share the same key. The first of these messengers which reaches a host leaves a kind of signature in the host by defining the key in the global store. This signature will be checked by the other messengers to determine if they are duplicates. The first messenger inserts itself in the `pqueue` queue and waits until it reaches the head of the queue. At this point, the messenger can find that it has reached the semaphore owner. If this is indeed the case, the messenger updates the semaphore data structures and `submit`s an `ackP` messenger to its origin. Otherwise, the messenger has not reached the semaphore owner, it moves itself to the probable semaphore owner and indicates that it is trying to acquire the semaphore ownership for its origin host by updating the `acqhost` entry in the semaphore. Subsequent duplicates of the messenger will find the signature left by the first messenger and will then act accordingly.

```
msgrP(ori, qack, queue, key, handle, src)
{
  d = getsemadict()
  if (known(d, handle)) {
    sema = get(d, handle)
    enterqueue(sema.accessqueue)
    if (sema.acqhost == src) { #duplicate msgr
      if (sema.owner != THISHOST)
        submit(sema.owner, msgrP(ori, qack,
          queue, key, handle, src))
      leave()
    }
    else {
      if (sema.owner == src) { #duplicate msgr
        leave()
        val = get(globdict, key)
        submit(origin(), ackP(ori, val, qack,
          queue, key, handle, src))
      }
      else {
        leave()
        if (!known(globdict, key)) {
          enterqueue(sema.acqhostqueue)
          if (!known(globdict, key)) {
            define(globdict, ori, origin())
            define(globdict, key, HOSTID)
            enterqueue(sema.pqueue)
            stopqueue(sema.pqueue)
            enterqueue(sema.accessqueue)
            if (sema.owner == THISHOST) {
              # at destination
              sema.owner = src
              sema.state = REMOTE
              sema.acqhost = NULL
              leave()
              startqueue(sema.pqueue)
              submit(origin(), ackP(ori, OK,
                qack, queue, key, handle,
                src))
            }
            else {
              if (sema.acqhost == NULL) {
                sema.acqhost = src
                dest = sema.owner
```

```
                  }
                  else {
                     if (sema.acqhost == src)
                        dest = sema.owner
                     else
                        dest = sema.acqhost
                  }
                  undef(globdict, key)
                  leave()
                  submit(dest, msgrP(ori, val,
                     qack, queue, key, handle,
                     src))
                  startqueue(sema.pqueue)
               }
            }
            else
               leave()
         }
      }
   }
   }
   else { # send back a nack
      submit(origin(), ackP(ori, ERROR, qack,
         queue,
         key, handle, src))
   }
}
```

As for the msgrP messengers, all copies of the same ackP messenger synchronize their execution through a common key. The messenger updates the semaphore data if it is not a duplicate, and if it has reached the final host, it unblocks the msgrP waiting for the semaphore ownership, otherwise it submits a copy of itself towards the final host.

```
ackP(ori, val, qack, queue, key, handle, src)
{
   enterqueue(qack)
   mykey = get(globdict, key)
   if (mykey == HOSTID) { # first ack
      define(globdict, key, val)
      stopqueue(qack)
      sema = getsema(handle)
      enterqueue(sema.accessqueue)
      sema.acqhost = NULL
      sema.owner = src
      leave()
      if (src == THISHOST)
         startqueue(queue)
      startqueue(qack)
   }
   if (src != THISHOST)
      submit(ori, ackP(val, qack, queue, key,
         handle, src))
   leave()
}
```

3.4 The V Procedure in Detail

The V procedure is used to relinquish a semaphore. If there is no local data for the semaphore, one is created by the messenger before proceeding. For a semaphore owned by the local host, (state is LOCKED or FREE), the semaphore pqueue queue is unblocked and the result of the operation stored in the global store. For a REMOTE semaphore, the calling messenger submits a msgrV messenger to the semaphore owner to relinquish the semaphore (see the remoteP procedure), and waits for the result of the operation. The messenger will be unblocked by the ackV messenger after the operation has been carried out. The V procedure is similar to the P procedure except that the messenger executing the V procedure never enters the pqueue queue and that it is a msgrV messenger which is submitted to release a remote semaphore. Therefore the code of this procedure is not shown here.

The remoteV procedure is similar to the remoteP procedure described above.

When a msgrV messenger reaches the semaphore owner, it unblocks the semaphore pqueue queue so that messengers trying to acquire the semaphore can do so and sends back to its origin the ackV messenger which convoys the result of the operation. And when it reaches a host which is not the semaphore owner, it forwards a copy of itself towards the semaphore owner.

```
msgrV(queue, key, handle, src)
{
   d = getsemadict()
   if (!  known(d, handle))
      submit(src, ackV(queue, key, ERROR))
   else {
      sema = get(d, handle)
      enterqueue(sema.accessqueue)
      if (sema.owner == THISHOST) {
         sema.state = FREE
         startqueue(sema.pqueue)
         leave()
         submit(src, ackV(queue, key, OK))
      }
      else {
         submit(sema.owner, msgrV(queue, key,
            handle, src))
         leave()
      }
   }
}
```

The ackV messenger is sent from the semaphore owner to the host where the messenger wishing to relinquish the semaphore is executing. This messenger is blocked in a queue waiting for the result of the operation. The ackV messenger just stores the result in global store and unblocks the queue to allow the waiting messenger to proceed its execution.

4 Discussion

4.1 Some Observations

The protocol ensures a kind of weak fairness in the sense that when messengers executing on the same host compete to acquire a semaphore, the first to enter the semaphore queue (pqueue) will get the semaphore; and when the messengers execute on different hosts, the first to reach the semaphore owner will get the semaphore. However, the protocol does not ensure that the first messenger to execute the `P` procedure will reach first the semaphore owner. This means that starvation; i.e, a situation where a messenger waits arbitrary long to acquire the semaphore can result.

Let us consider the case with two hosts h_i and h_j with $(h_i \, \mathcal{R}^s_{\texttt{own}} \, h_i)$ and $(h_j \, \mathcal{R}^s_{\texttt{own}} \, h_i)$. On h_j, messenger m_j tries to acquire the semaphore and `submit`s an `msgrP` messenger $msgrp_j$ to the semaphore probable owner; i.e, h_i. The $msgrp_j$ messenger reaches host h_i, and acquires the semaphore ownership, updating the ownership relation to reflect this fact. We have now $(h_i \, \mathcal{R}^s_{\texttt{own}} \, h_j)$ and $(h_j \, \mathcal{R}^s_{\texttt{own}} \, h_i)$ which is clearly a loop. Until an `ackP` messenger will reach host h_j to confirm semaphore ownership acquisition, there is no true owner of the semaphore. If a `V` operation is performed at this time, it can result in a lot of messengers being exchanged between the two hosts. Indeed, a `msgrV` messenger reaching host h_i will find that the probable semaphore owner is h_j and therefore will `submit` another `msgrV` to release the semaphore on host h_i. But, this last messenger will find that on host h_i, the semaphore probable owner is h_j and will at its turn `submit` a copy of itself to h_j and so on. However this is only a temporary situation which disappears when a `ackP` messenger from h_i reaches h_j and updates there the probable semaphore owner.

Before a messenger can actually acquire a semaphore, the host where it is executing must be the semaphore owner. Now if a number of messengers executing on two different hosts are competing for a semaphore, one can imagine a situation where the semaphore ownership oscillate quickly between the two hosts because each `P` operation results in network activity. This behavior is similar to thrashing which has been identified in some distributed memory systems. As for thrashing, network traffic between the two hosts can be considerably reduced by letting each host retain the semaphore ownership for at least a given amount of time. By retaining the semaphore ownership longer on a host, messengers executing there can have a better chance to acquire directly the semaphore; i.e, without any network traffic being generated.

The choice of the timeout value can influence the amount of network traffic generated. This problem is common to all protocols which must ensure reliability using a retransmission mechanism. A too big timeout value can result in a loss of efficiency in the situation of messenger loss because a messenger waits a long time before it can notice messenger losses. On the contrary, a too small timeout value can result in unneeded duplicate messengers being generated. The choice of this timeout value is not easy. Ideally this value should be dynamically adjusted according to the network load. In our implementation, we have used the simple solution of incrementing exponentially the timeout value after a timeout.

An optimization of the `P` procedure is to bypass a number of hosts belonging to the sequence of probable owners of the semaphore. Indeed, when a `msgrP` messenger trying to acquire the semaphore ownership for host h_i reaches a host h_j, it updates the `acqhost` field in the semaphore on this host. All subsequent `msgrP` messengers reaching h_j will bypass some hosts in the ownership sequence by moving directly to h_i which is now a probable owner of the semaphore closer to the true semaphore owner.

4.2 LIMITATIONS

The protocol presented in this paper works fine in a static environment. It does not handle either the case where hosts are added/removed to the system or the situation of failure. However, with the service architecture presented in this paper, the protocol can be adapted to handle dynamicity. All the complexity for managing up/down dynamicity can be handled by the `up` and `down` procedures.

Moreover, the protocol does not handle routing of messengers. It assumes a kind of single ETHERNET where each platform can be reached directly from any other. However the view of single ETHERNET can be achieved even in the case of multiple linked networks with an appropriate routing function implemented inside "traveling" messengers.

The protocol implements a distributed binary semaphore. We show below how it can be adapted and extended to handle the more general non binary semaphores.

4.3 CORRECTNESS

We give here an informal proof of the correctness of our protocol. To assess the correctness of the given protocol, it is essential to show that (a) semaphore ownership is maintained consistently; i.e, at any time there cannot be more than one true owner of a semaphore and (b) that the true owner of a semaphore is always found. Given these two assumptions, it becomes simple to see that the protocol achieves the distributed semaphore service. Indeed, when the true semaphore owner is found, the semaphore ownership is transferred to the host where the messenger trying to acquire the semaphore is executing. Then, the messenger acquires the semaphore and blocks the semaphore `pqueue` queue where all messengers are inserted when they do the `P` operation. This ensures that only one messenger can acquire the semaphore. And when a `V` operation is performed, the true semaphore owner is found and the semaphore `pqueue` is restarted freeing effectively the semaphore.

4.3.1 Semaphore Ownership is Consistent

Here, we must show that a semaphore cannot be owned by more than one host. This follows from the way the semaphore ownership is updated.

Definition 4.1 *The semaphore ownership for semaphore s is consistent if $\forall h_i, h_j \in \mathcal{H}$ with $(h_i \, \mathcal{R}^s_{\text{own}} \, h_i)$ and $(h_j \, \mathcal{R}^s_{\text{own}} \, h_j)$ then $h_i = h_j$; it means there cannot be more than one true owner for a semaphore.*

At semaphore creation time and whenever a host creates local data for a semaphore, the probable owner of the semaphore is initialized to be the host where the semaphore has been created. This ensures that at creation time semaphore ownership is consistent and that whenever local data is created for a semaphore, if semaphore ownership is consistent, it remains consistent. Indeed, creating local data for a semaphore only adds a new element of the the type $(h_i \, \mathcal{R}^s_{\text{own}} \, h_j)$ (with h_j the host where the semaphore was created, and h_i the host where local data is being created for the semaphore) to $\mathcal{R}^s_{\text{own}}$ relation. And semaphore ownership is updated only when the true owner of a semaphore is found. Firstly the probable owner of the semaphore is updated on the true owner of the semaphore. This changes the unique element of type $(h_j \, \mathcal{R}^s_{\text{own}} \, h_j)$ of the $\mathcal{R}^s_{\text{own}}$ relation to an element of type $(h_j \, \mathcal{R}^s_{\text{own}} \, h_i)$. Secondly the ownership is updated in reverse order on all the hosts belonging to the sequence of hosts which have been visited while searching for the semaphore owner. Elements of $(h_i \, \mathcal{R}^s_{\text{own}} \, h_j)$ type are updated but remain of the same type. And finally, update is done on the new semaphore true owner, changing an element of $(h_i \, \mathcal{R}^s_{\text{own}} \, h_j)$ to one of $(h_i \, \mathcal{R}^s_{\text{own}} \, h_i)$ type. Hence semaphore ownership is updated consistently.

4.3.2 Semaphore Owner is Always Found

The owner of a semaphore changes dynamically with the requests of messengers to acquire the semaphore. The owner of a semaphore can change while a messenger is searching for it. This can occur only when messengers are competing to acquire the semaphore (execute the `P` procedure concurrently). The case where only one messenger is executing the `P` procedure is simple. Since semaphore ownership is consistent, the semaphore owner will always be found. When two messengers are executing the `P` procedure concurrently, two cases can arise. In the first case, the two messengers can follow different sequences of hosts of the $\mathcal{R}^s_{\text{own}}$ relation and reach the semaphore owner. They then serialize their actions by inserting themselves in the `pqueue` queue. The first messenger to reach the head of the queue will acquire the semaphore and update the semaphore ownership. When the second messenger comes at the head of the queue, it finds that the host is no longer the semaphore owner and will continue its way to the semaphore owner. In the second case, the sequences of hosts followed by the messengers intersect before reaching the semaphore owner. The first messenger to reach the host h_j where the sequences intersect will update the `acqhost` value on this host, indicating that it is trying to acquire the ownership of the semaphore for a host h_i. The messenger will then continue on its way towards the semaphore owner, following the sequence of hosts defined by the probable owner relation. When the second messenger reaches the host h_j, it finds that another messenger originated from host h_i is trying to acquire the same semaphore. The messenger will then bypass the remaining sequence of probable owners and will proceed directly on host h_i where it will wait to acquire semaphore ownership after it has been acquired by h_i.

4.4 Implementation

The protocol presented in this paper has been implemented in the `MØ` messenger environment. The different messenger platforms were linked by UDP channels. In our implementation, each messenger `submit`ted to a remote host carries with it its code (all the messengers fit in one UDP packet and do not have to be fragmented). An alternative implementation would be to store the code of the messengers in each platform; when a messenger comes in a platform it first finds the code to execute.

The `P` and `V` operations are implemented as procedures executed in the scope of the calling messenger. They are accessible to all messengers but they protect the data on which they work; i.e., messengers cannot directly access the semaphore data or the process queues used to synchronize the execution of the messengers generated by the `P` and `V` procedures.

4.5 Extension to Semaphores with N Entries

For a semaphore with N entries, a value field must be maintained. Our binary semaphore can be extended to a semaphore with N entries by adding a value field to each local data representing a semaphore and by maintaining consistency of this value among all the local values of the semaphore with the P and V procedures.

One straightforward way for maintaining the semaphore value consistency is to have only one copy of the semaphore value be meaningful, for example that on the true semaphore owner. As for our binary semaphore, the ownership of the semaphore is requested before acquiring the semaphore and the value field updated. And when releasing the semaphore, the `value` field is updated only on the true semaphore owner.

Another approach where each local value represents the true value of the semaphore is possible. However, each P and V procedure must be made visible to all hosts to allow them to update the value of the semaphore. This is clearly not efficient since it generates a lot of network traffic.

5 Summary

Mobile code is now considered as an alternative way for the implementation of distributed applications. The messenger environment presented in this paper allows the exchange of programs called messengers between hosts linked by unreliable channels. Flexibility is achieved by letting messenger execution environments (messenger platforms) provide local services only. Inter-node services are implemented at a high level with messengers instead of having hard-wired protocols in software. Code becomes mobile and replaces the classical client/server programming model. A service architecture based on a set of uniform conventions shared between all messenger platforms and more suited for mobile code has been proposed, and the distributed semaphore service has been presented and implemented in that framework. Work remains to be done to find a uniform and expressive way for describing the interface for an appropriate usage of a service.

References

[1] M. ACCETTA, R. BARON, W. BOLOSKY, D. GOLUB, R. RASHID, A. TEVANIAN, and M. YOUNG. *Mach: A New Kernel Foundation for Unix Development*. In Summer Conference. Usenix Association, 1986.

[2] M. BETZ. *OMG's CORBA*. Dr. Dobb's Special Report, (225):8–12, Winter 1994/1995.

[3] E. W. DIJKSTRA. *Cooperating Sequential Processes*. Technical report, Technological University, Eindhoven, 1965.

[4] E. W. DIJKSTRA. *Hierarchical Ordering of Sequential Processes*. Acta Informatica, 1(2):115–138, 1971.

[5] G. DI MARZO, M. MUHUGUSA, C. TSCHUDIN, and J. HARMS. *The Messenger Paradigm and its Impact on Distributed Systems*. In ICC'95 workshop on Intelligent Computer Communication, 1995.

[6] A. N. HABERMANN. *Synchronisation of Communicating Processes*. Communication of the ACM, 15(3):171–176, 1972.

[7] Adobe Systems Incorporated. *PostScript Language Reference Manual*. Addison Wesley, 1991.

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

[9] OSF. *Introduction to OSF DCE*. Prentice Hall, 1992.

[10] M. ROZIER, V ABROSSIMOV, F. ARMAND, I. BOULE, M. GIEN, M. GUILLEMONT, F. HERRMANN, C. KAISER, S. LANGLOIS, P. LEONARD, and W. NEUHAUSER. *Overview of the CHORUS Distributed Operating System*. Technical Report CS/TR-90-25, Chorus Systèmes, 1990.

[11] A. S. TANENBAUM, M. F. KAASHOEK, R. van RENESSE, and H. BAL. *The Amoeba Distributed Operating System-A Status Report*. Computer Communications, 14:324–335, July/August 1991.

[12] C. F. TSCHUDIN. *On the Structuring of Computer Communications*. PhD thesis, Université de Genève, 1993. Thèse No 2632.

[13] C. F. TSCHUDIN. *An Introduction to the M0 Messenger Language*. Technical Report No 86 (Cahier du CUI), University of Geneva, 1994.

[14] C. F. TSCHUDIN, G. DI MARZO, M. MUHUGUSA, and J. HARMS. *Messenger-based Operating Systems*. Technical Report No 90 (Cahier du CUI), University of Geneva, 1994.

[15] R. LINO VALVERDE. *MSGR-S: Un environnement d'exécution de messagers basé sur un interpréteur Scheme parallèle*. Diploma thesis, University of Geneva, 1994.