A review of four distribution infrastructures

To cite this article: J O'Connell et al 1994 Distrib. Syst. Engng. 1 202

View the article online for updates and enhancements.

Related content

- Java transactions for the Internet
  M C Little and S K Shrivastava

- DC++: distributed object-oriented system support on top of OSF DCE
  A B Schill and M U Mock

- Object replacement using dynamic proxy updates
  S Menon and R J LeBlanc Jr
A review of four distribution infrastructures

John O'Connell, Nigel Edwards and Robert Cole
Hewlett-Packard Laboratories, Filton Road, Stoke Gifford, Bristol BS12 6QZ, UK

Received 2 September 1993

Abstract. This paper describes our experience in designing and implementing a
group-ware application (a shared whiteboard) on four different distribution
infrastructures. We describe our application and how its design was influenced by
reliability requirements. We review the four platforms and discuss the useful, and
not so useful, features of each platform.

1. Introduction

Our group is working on a long term research project to
improve the support for reliability provided by distribution
infrastructures and the abstraction of reliability as presented
to an application programmer. We are currently
investigating an interaction paradigm called 'groups' [1]
and its use in supporting reliability. We chose a shared
whiteboard application to explore the use of groups and the
support required for reliability in such applications.

This paper describes the design and implementation of
the shared whiteboard application on four different
distribution infrastructures:

- ANSAware 3.0 [2], a prototype system produced by the
  ISA ESPRIT project,
- the HP DOMF, a prototype of the HP product which
  implements the OMG Common Object Request
  Architecture [3],
- ISIS [4] (version 2.1), the process group communication
  system from Cornell University.

In addition we developed a design, but no implementation,
for
- Arjuna [5], a toolkit for constructing reliable distributed
  systems in C++ from Newcastle University.

The paper first describes the design of the application. The
design is compared with other whiteboard applications
reported in the literature, especially with regard to our use
of the groups paradigm. The two main sections of the paper
describe the four infrastructures we used and compare the
features that we used in our application. Finally we draw on
our experience for lessons about distribution infrastructures
and for future work.

2. The context and the design of the whiteboard

The whiteboard application was chosen because it enabled
us to explore the use of groups [1]. A group is a set of
objects which can be treated as a single object by a user of
the group. Messages sent to the group are delivered to all
members of the group or none at all (atomic delivery). Messages from different users are delivered in the same
order so that each member of the group processes the
messages in the same sequence. The user of the group is
unaware of the size of the group or the nature of the
members; thus the group is transparent to the user. The core
mechanism of a group is the use of multicast communication
from the group user to the group members. A member of a
group may send messages to the group as well.

The whiteboard application stresses the infrastructure in
its requirement for response to a human user [6]. The
whiteboard could be used for scribbling, brainstorming,
recording notes and decisions, etc. The whiteboard supports
WYSIWIS (what you see is what I see) which requires each
person viewing the same whiteboard to

3. Application overview

This section gives an overview of the application design
which provides the basis of each implementation. The
application has two parts: establishing a conversation, and maintaining the conversation. The parts are represented by independent application objects. The register object has two functions:

1. enabling a user at a workstation to create a conversation through a graphical user interface,

2. registering the workstation user as a potential participant for conversations.

To create a conversation the user invokes the user interface of his local register object, which would normally be permanently running on the workstation. The conversation creator is given a list of all possible participants, which represents the current set of active register objects in his domain. When a list of proposed participants has been drawn up the user asks for the conversation to be created. The user's register object then asks the register object representing each named participant to create a second object, the whiteboard object, on the participant's local workstation. Before creating a whiteboard object on a potential participant's workstation a register object pops up a window asking the user if he wants to join the conversation, with a time-out. Acceptance of the invitation causes the register object to create the whiteboard object. Refusal, or time out, causes the register object to return a negative reply. When all of the called register objects have replied the creating register object creates a whiteboard object for the initiating user and produces a list of participants which is sent to each whiteboard object.

The set of whiteboard objects now represent the conversation and are independent of the register objects. They each have a copy of the membership list, and it is the maintenance of this membership list that causes the most difficulties for the infrastructure. Each whiteboard object displays a whiteboard window which allows the user to type, scrawl using coloured pens, airbrush, erase, or read in bitmaps and saved images. Each change to the user's whiteboard is multicast to each member of the conversation so that each whiteboard is a copy of every other whiteboard. During the conversation other users may be invited to join and existing participants may leave. All changes to the membership list are multicast to the whole conversation. The process is summarized in figure 1.

2.2 Non-features

The whiteboard is intended to be one of several communication channels between the participants so there were two things we did not tackle. We did not provide any floor control for deciding who could change the whiteboard; anyone can change the image at anytime. With this potential concurrency, we did not attempt to order the changes, so it is possible for members to see the changes to the same location on the whiteboard in a different order.

New participants into an existing conversation can be added to the participants list, as can existing members be removed. A new user currently obtains the state of the whiteboard through a backdoor channel. An existing participant can be asked to save their whiteboard state to a file; the new user then loads that file (via an out-of-band file transfer if necessary).

2.3 Novelty of the design

Our design for the whiteboard, based on groups, is sufficiently different from those reported in the literature that it can be considered as an alternative programming paradigm for distribution. The best way to describe our design is to compare it with other designs. In their paper [8] Levy and Tempero describe two implementations of a whiteboard application and suggest that these represent different programming models: client-server and object-oriented. We believe we have both object-orientation and client-server aspects in our design.

The first model described by Levy and Tempero is what they call client-server. The client contains a copy of the whiteboard state and can send updates to the server when the client's user changes the local representation. The client can also receive changes from the server and display them to the user. The server is at a fixed location and contains all of the remaining application state: the whiteboard state and the list of clients. A server contains the state of a number of whiteboards. (Levy and Tempero also include a list of whiteboards as part of the state, as our application does not use this we have not included it in this description). Figure 2(a) shows the design.

The second model described by Levy and Tempero is what they call object-oriented. This is essentially the same

![Figure 1. Summary of application design.](image-url)
functional design except that the server object is location independent and can migrate around the distributed system. Each server object contains the state of one whiteboard, see figure 2(b). In both models there is a single master copy of the application information (the whiteboard state) and each client interacts directly with the object holding this information. This only requires point-to-point communication between the whiteboard object and each client.

In our application, each client has a copy of the whiteboard state which is treated as a replica. Whenever the local copy is changed by the user, the whiteboard object sends the changes to every other whiteboard object. In this sense each object issuing a change is treating the other objects as servers for the purpose of the interaction. Thus the client-server relationship is dynamic and depends on the application usage. In the first two models it appears that the whiteboard state is the key information of the application, since there is only one master copy. In our design, the key information is the list of participating whiteboard objects, see figure 3. We try very hard to maintain the integrity of this list over the replicas.

The availability of groups has enabled us to utilize a different approach to the design of the whiteboard application. We have eliminated a single point of failure in the application (the master state).

3. Review of the infrastructures

This section contains a brief description of each of the four platforms we used. The description concentrates on those aspects of the platform that are used in the comparison discussion in the next section.

3.1 ANSAware

ANSAware [2] is an implementation of the ANSA architecture [9]. The computational model supported is described in [10]. The basic building block of ANSA is that of a service. Objects which use a service are called clients; objects which provide a service are called servers. A server object has a number of interfaces, which contain operations that can be invoked by other objects; each interface provides a service. Objects interact using RPC semantics.

In our application, each client has a copy of the whiteboard state which is treated as a replica. Whenever the local copy is changed by the user, the whiteboard object sends the changes to every other whiteboard object. In this sense each object issuing a change is treating the other objects as servers for the purpose of the interaction. Thus the client-server relationship is dynamic and depends on the application usage. In the first two models it appears that the whiteboard state is the key information of the application, since there is only one master copy. In our design, the key information is the list of participating whiteboard objects, see figure 3. We try very hard to maintain the integrity of this list over the replicas.

The availability of groups has enabled us to utilize a different approach to the design of the whiteboard application. We have eliminated a single point of failure in the application (the master state).
describe some specific characteristics of a service instance. Clients can request a service of a particular type, in which case the trader will arbitrarily select and return one registered interface of that service type. To further constrain the trader's search clients can provide constraint expressions which define the attributes of the service they want. The trader is the only well known interface in the system. All other interfaces are registered in the trader, including those of the other architectural services.

Factories provide the architectural service for instantiating objects. Factories can only instantiate objects on the node on which they reside. Each one of the attributes which a factory registers with the trader is the name of the node on which it is running. The third architectural service we used is the notification service. If an object is interested in the demise of another object it can register this with the notification service. Should the registered object die for some reason, the notification service will inform every one who registered an interest in it. This is useful if one wishes to avoid invoking operations on non-existing objects. The notification service requires a daemon to be running on each node, so it can cope only with (Unix) process failures rather than node failures.

ANSAware supports multiple threads and provides event counts and sequencers [11] for synchronization between threads. We used these primitives to build locks and semaphores.

3.2 DOMF

The DOMF is an early prototype system from Hewlett-Packard which implements the first OMG Common Object Request Broker Architecture (CORBA) [3]; this version predates the later additions. The CORBA defines a standard for inter-object communication principally through an interface definition language. The DOMF provides additional facilities for identifying objects, creating and destroying objects, installing classes and handling objects' persistent data.

Objects in this system are instances of classes. All objects of the same class support the same external interface. Once a class has been installed on a machine, instances of that class can be created on the machine. Objects can be created and destroyed dynamically. The system also provides a persistent storage mechanism—when an object is created, it can also be assigned a storage area in which to store its persistent data. It is the responsibility of each object to manage its own persistent data, the system merely provides the storage space for the data.

The DOMF views a distributed application as a collection of communicating objects. Objects communicate by issuing requests, with RPC semantics. If the destination object is not running when the request arrives at its machine, the DOMF activates the objects (assigns it a process). Active objects may be de-activated by the system if there are too many objects running. If the operation completes successfully the source object is informed and any output parameters are returned. Otherwise the source object is informed that an error has occurred. All requests in the current DOMF prototype generate a reply, although CORBA allows both synchronous and asynchronous interaction. CORBA also defines an Interface Definition Language [3] for defining class interfaces.

An object must know the identifier (object reference) of another object to communicate with it. Objects are assigned unique identifiers when they are created. Other identifiers can be passed as parameters and can be stored persistently.

The DOMF imposes tight control over object creation. The system must be explicitly asked to create the object. Creating an object includes assigning it an object identifier and, if requested, allocating storage space for the object's persistent data. Any DOMF object can ask the system to create another object locally. This request must specify a class identifier, and the DOMF will only comply with the request if the class with that class identifier has been installed locally. Classes are installed by the system administrator on a per-machine basis. Classes also have unique identifiers—the system administrator is responsible for generating a class identifier before installing it.

The system must also be explicitly asked to destroy objects. Killing the process in which the object is executing is not sufficient because from the DOMF's perspective this is equivalent to de-activating the object. If the DOMF is asked to destroy an object, it removes all references to the object from the system (and hence no further messages will be delivered to the object) and deletes the object's persistent store, if it exists. The DOMF, however, is not responsible for killing the process in which the object is active.

3.3 ISIS

ISIS [12] is a system developed at Cornell University which supports distributed process groups. It allows processes to create groups and to join and leave groups. It also provides a set of multicast protocols which allow clients of a group to communicate with all the members of the group. Different protocols provide different ordering guarantees on the messages (for example, one protocol gives a total ordering, another preserves causality).

Groups of processes are identified by name. It is not possible to simultaneously have two groups with the same name, but group names need not be unique in time. To join a group, a process specifies the group's name. If a group of that name does not exist, then a new group is created. Otherwise the process is added to the group and all the members are updated with an updated membership list. A process can belong to more than one group.

The group ceases to exist when the last member leaves. Processes may explicitly leave a group, or alternatively ISIS may detect that the process has terminated. A group member can also ask that the group be deleted, in which case the group also ceases to exist.

When a process joins a group, it must specify the routine which is to be invoked when the group membership changes. ISIS passes the latest group membership list, as well as the address of any member which has joined or left, as arguments to this routine. It is also possible for a new group member to have its state initialized by an existing group member.
Multicast message can be addressed to a group, to a list of processes or to an individual process. To multicast to a group, a client addresses the message to the group address. This can be obtained from ISIS by specifying the group name. A group's address does not change during the lifetime of the group (and thus may be shared and re-used).

Processes do not accept arbitrary messages. Instead a process provides a different routine to handle each type of message which it expects (when a process starts up, prior, for example, to joining any groups, it must declare which routine handles each message type). When a message arrives for a process, ISIS invokes the appropriate routine.

The sender can pass arguments to, and receive return values from, the recipients of the message. The sender also specifies the number of replies which it requires (the options include 'ALL', 'MAJORITY' and 0)—the client may, for example, be only interested in the first response received (although of course the message will still be delivered to all the destinations identified in the invocation). The thread which does the invocation suspends until the required number of responses have been received.

3.4 Arjuna

Arjuna [5] is a toolkit for constructing reliable, distributed object-oriented systems being developed at the University of Newcastle upon Tyne. It consists of three main components: core Arjuna, the stub generator and the remote procedure call (RPC) mechanism. The computational model is one of distributed objects whose methods are invoked using RPC. Core Arjuna is a C++ class library which uses inheritance to provide concurrency control, persistence and recovery. Atomic actions are provided to coordinate these mechanisms.

Creating objects in Arjuna is very easy: one just invokes the C++ constructor for the class and the object is created by the infrastructure. Objects are distinguished by class and unique identifier (uid). In Arjuna the only way of assigning a uid to an object is to make it persistent. Persistent objects are assigned unique identifiers and are stored in the object store. Any object which is derived from the Arjuna class StateManager can be declared to be persistent.

Concurrency control is provided by read and write-locks. Any object derived from the Arjuna class LockManager inherits operations to set and manipulate read and write-locks which enforce multiple-readers-single-writer [13].

Arjuna provides roll-back state-based recovery. This means that the state of an object is stored so that, should a failure occur during an operation or sequence of operations, the old state is recovered [14].

One of the philosophies behind Arjuna is that distribution should be transparent; the interface to an object is precisely defined by its non-distributed implementation. Hence in Arjuna stubs are generated from C++ class definitions, not from a separate interface definition written in an IDL. This means that the programmer uses C++ as the interface definition language.

The RPC mechanism used in Arjuna is Rajdoot [15]. Rajdoot is designed for general purpose use, so it is not language specific.

Arjuna tries very hard to hide distribution from the programmer—the syntax for remote and local method invocation is identical. A distributed Arjuna program consists of one or more driver programs which act as clients invoking operations on a number of remote server objects. Some server objects may themselves be clients of other, possible remote, server objects. Every time a remote (server) object S comes into scope in a client C, C invokes the C++ constructor for S's client stub object. This requests the RPC manager process running on the remote node to create a server process which runs the remote object's methods.

We completed a detailed design of the Arjuna version but did not implement the application because of hardware compatibility problems.

4. Comparison of implementations

The three implementations and the Arjuna design all use the same user interface code so to the user each implementation is functionally equivalent. In all of the designs we model a conversation as a group of objects where the failure of one object does not affect the other conversation participants. However radical differences between the designs appear if we examine their internal structure and if we distinguish between the support provided by the underlying platform and the work done by the application programmer. The purpose of this section is to detail these differences.

Although ISIS does not support the notion of an object, for design purposes, we considered a single process in ISIS to be an object. Such a process, corresponding to a group member, sends and receives messages, has an identity, and needs to react to changes in its environment, such as the failure of a process or a machine. Thus we will use the word object loosely to include individual ISIS processes. The remainder of this section compares the features of the distribution platforms that we used with the whiteboard application. The features are: the object lifecycle, finding object instances, multicast communication, concurrency, group membership, failures, exception handling, and programming support.

A performance comparison of the infrastructures could have been made using the common application as a basis. However there are two reasons why numerical results would be unfair and unfortunate. First the ARJUNA and ISIS systems were developed in an academic environment to explore concepts, not as production systems; the ANSA and DOMF systems were not intended as products which could be subject to this rigour in public. Second, there are too many differences between the infrastructures and the amount of work done by the application to make a simple comparison of each infrastructure possible. The origins of each of these infrastructures were very helpful in developing the application and in the preparation of this paper—we have no wish to subject them to simple and potentially unfair comparison.
4.1 Object lifecycle

Table 1 summarizes the differences between the platforms on issues relating to object lifecycle. The DOMF imposes the tightest control over object creation and deletion. The system must be explicitly asked to create and destroy objects. This means that creating the initial set of register objects in our application is not trivial. Client programs had to be written to start the application. These client programs are not viewed as objects by the system—they can only issue requests. Every user of the application has to run a client program which asks the DOMF to create his/her register object. Creating new whiteboard objects is easier—register objects ask the DOMF directly. In both cases, the new object is not activated until a message is received by the DOMF for it.

With ANSAware, register objects are created directly by the user by running a process containing the object code as an ordinary command (activating an object is equivalent to creating it because there is no notion of persistent objects). When a new whiteboard object is required, the register object asks the local factory to create it. The register object must supply the name of the new object’s class with this request. Factories can also be asked to terminate an instantiated object. In ANSAware, killing the process in which the object is active is equivalent to destroying the object.

In the ISIS version, register objects are started directly by the users, and whiteboard objects are created using the standard fork and exec routines. One disadvantage of this approach is that the register object has no identifier or address for any whiteboard object which it creates. ISIS does provide a facility for starting new objects on a (possibly remote) machine which yields the address of the new object. We decided not to use this facility because register objects only create whiteboard objects locally and when a whiteboard object does join a conversation group, the other participants obtain its address from the ISIS infrastructure, so it is not necessary for the inviting object to multicast that information (as happens in the other implementations). Objects in ISIS are not persistent.

Objects in Arjuna are created by invoking the C++ constructor for the class. Both register objects and whiteboard objects are declared to be persistent so that the system assigns unique identifiers to them, but we make no other use of Arjuna’s facilities for persistence. These objects must be explicitly destroyed because they are persistent.

<table>
<thead>
<tr>
<th>Table 1. Object lifecycle issues.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Does the system support</td>
</tr>
<tr>
<td>object creation?</td>
</tr>
<tr>
<td>Can objects be created</td>
</tr>
<tr>
<td>externally?</td>
</tr>
<tr>
<td>Are objects activated</td>
</tr>
<tr>
<td>when created?</td>
</tr>
<tr>
<td>Are objects persistent?</td>
</tr>
<tr>
<td>Must objects be explicitly</td>
</tr>
<tr>
<td>destroyed?</td>
</tr>
</tbody>
</table>

4.2 Finding object instances

A major issue in designing distributed applications is determining how the object instances discover each other’s existence, and also, each other’s identity. For example, in our application, both register objects and whiteboard objects need to find register objects representing other users.

ANSAware provides a general trading service, described in section 3.1. Every register object registers itself with the trader and supplies its user’s name as an attribute. This allows the other objects in the application to get a mapping from user name to register object identifier.

In the DOMF we implemented a simple trading object to provide the minimum trading functionality for our application. The register objects have to agree on which trader to use. Thus before any register object can be created, a trader object has to be created. A similar approach was adopted in the Arjuna design.

In the case of ISIS, the register objects form a group with a well known name, other objects multicast to this group to obtain the required information. This was the sole reason for putting the register objects into a group. No other messages are multicast to this group, even messages concerning group membership changes are ignored. This is not a general solution to the problem of finding other objects and services, but it worked well in our application.

4.3 Multicasting

In our application, each whiteboard object is responsible for sending changes performed by the local user to the other whiteboard objects in the conversation. In the case of the DOMF, ANSAware and Arjuna this required simulating multicast on top of RPC-based communication primitives. In contrast, ISIS provides a rich set of multicast primitives.

We decided that maintaining consistency at the application level was not a priority. Participants in a conversation are allowed to have slightly different views of the whiteboard. It would have been possible in ISIS to ensure consistent views if we had used one of the stronger multicast primitives, but this would have adversely affected performance; in particular, local updates to the display would have had to be delayed.

The DOMF and ANSAware implementations both use threads to simulate multicast. When a whiteboard object wishes to send an update to the other participants, it creates a new thread for each destination. These threads perform the remote invocations simultaneously, and the main thread waits for all the forked threads to complete. The version of Arjuna which we used does not support threads, so multicast is simulated by a sequence of RPCs.

The multicasting of invitations to would-be conversation participants caused an additional problem because the reply depends on the users’ response, and we needed to avoid an infinite delay. In the DOMF and ANSAware, timeouts are used so that, if the remote user does not respond to the invitation within a specified period, it is assumed that the invitation had been rejected. In ISIS the register object just waits for an acknowledgment that
the invitation has been received, and then proceeds to create a new ISIS group for the conversation participants. Whenever an invitee decides to accept the invitation, a whiteboard object is created and it joins the group. This approach could not be used in the DOMF or ANSAware because conversation groups are not named and so an object outside of a group cannot request to join. The Arjuna design adopted a similar approach to the ISIS implementation, with invitations being acknowledged before the user makes a decision. However, if the user does accept the invitation, a ‘join conversation’ operation is invoked on the object which sent the original request.

4.4 Concurrency

Of the four systems only Arjuna and ISIS provided primitives to serialize distributed actions; ISIS through the totally ordered atomic multi-cast which was considered to be too expensive. We could have used atomic updates for each change to the whiteboard. This would have provided a consistent ordering and guarantee that each whiteboard was the same as any other. Atomic updates have a high overhead, approximately three times as much network traffic as non-atomic updates.

Given the expected frequency of updates, and the need for real-time response to the human user we decided that the performance was more important than consistency on the whiteboard. Changes to the membership of the conversation are relatively infrequent and it is important that each member of the conversation have a consistent view of the participants list: all changes to the group membership are atomic.

4.5 Group membership information

In all the versions a conversation is modelled as a group of objects, and there is information relating to this group which must be available to all the members, particularly the list of current participants. It is important that all members see the same participant’s list, otherwise some participants may not receive all updates. Each whiteboard object therefore maintains a table containing participants’ names, as well as the identifiers of their whiteboard objects.

Consistency for the table of participants means that each participant’s copy contains the same names as every other participant’s copy. Making changes to the copies therefore requires some guarantees: all the copies are updated, or none of them are; and the updates must occur in bounded time. The extent to which the infrastructures can support these guarantees affects the possibility that in one implementation the participants table may become inconsistent. The Arjuna and ISIS infrastructures are able to provide these guarantees, the ANSA and DOMF infrastructures are not.

In the DOMF, ANSAware and Arjuna versions, the task of maintaining the participants table is handled solely by the application. Changes in group membership are sent to all participants. In the DOMF and ANSAware implementations it is possible for inconsistencies to arise. For example, if two new participants attempt to join at the same time then, if the multicasting of the details of the new members overlap, the new members may not get the information on each other. ISIS prevents this by ordering group joins. Each member sees the details of the new participants in the same order, and so the multicasting of the details do not overlap. The Arjuna implementation solves the problem by making all group joins atomic actions.

ISIS does not however allow applications to modify the membership information held by the system—new conversation participants still have to multicast some details of themselves to the other participants. The application had to provide the routine which is invoked by the infrastructure when the group’s membership changes and the routine which is invoked by new conversation participants when multicasting details of themselves.

In the RPC-based systems, local updates to the whiteboard state are multicast to all the objects listed in the table of participants, so if inconsistent versions of the table exist, all participants may not receive the same updates. In contrast, in the ISIS version, the table of participants only exists so that users of the application can see who the other participants are, with ISIS providing a group address to which all updates are addressed. This guarantees that all participants receive the same updates, although not necessarily in the same order. The ISIS version only uses the object identifiers in the table for determining the names of failed participants (ISIS provides the identifier, the table supplies the matching name).

4.6 Interaction process and machine failures

One of the primary goals of our whiteboard application is to demonstrate reliability, so it is interesting to examine how the different infrastructure implementations cope with communication, process and machine failures. Failures can be detected at either the system or application level, and similarly can be handled by either (or they can be ignored, leading to a degradation in performance). Failures which are not handled by the infrastructure usually generate exceptions. Table 2 summarizes how the application level becomes aware of the different failure modes.

The next section will describe how the application handles exceptions raised by communication with other objects. Here we discuss how the set of whiteboard objects in a conversation use the facilities provided by the infrastructures for detecting and handling failures.

ANSAware provides a notification service for objects. In our application all the whiteboard objects in a particular conversation declare an interest in each other. If one object fails the other participants are informed. The service is not ideal: failure detection is not immediate and the notification service cannot handle machine failures. Whiteboard objects may also detect object failures by time-outs in invocations.

In ISIS, all the whiteboard objects in a conversation form a group. If ISIS detects the failure of an object, due to either process or machine failure, the other participants are informed. Failure detection in ISIS is very rapid, and ISIS also guarantees that all messages sent by a failed object prior to the failure are delivered before the failure notification message. This mechanism is so reliable that, when a participant decides to leave a conversation, the
A review of four distribution infrastructures

Table 2. How an application object becomes aware of a failure.

<table>
<thead>
<tr>
<th></th>
<th>ANSAware</th>
<th>DOMF</th>
<th>ISIS</th>
<th>Arjuna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process failure</td>
<td>RPC exception (time-out)/</td>
<td>Not detected by application</td>
<td>Group membership changes/Exception (insufficient replies)</td>
<td>Atomic action aborted/ RPC exception</td>
</tr>
<tr>
<td></td>
<td>Informed by Notification Service</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine failure</td>
<td>RPC exception (time-out)</td>
<td>RPC exception</td>
<td>Group membership changes/Exception (insufficient replies)</td>
<td>Atomic action aborted/ RPC exception</td>
</tr>
<tr>
<td>Communication failure</td>
<td>RPC exception (time-out)</td>
<td>RPC exception</td>
<td>Exception (insufficient replies)</td>
<td>Atomic action aborted/ RPC exception</td>
</tr>
</tbody>
</table>

whiteboard object just shuts itself down, relying on ISIS to detect its absence and to inform the other participants. ISIS runs a daemon on each machine which watches the status of processes and communicates with other daemons to maintain the status of the whole system.

In both these implementations, the application programmer must supply the routine which is invoked by the system when a failure notification is delivered. In both cases failed participants are removed from the list of participants.

The DOMF implementation adopts a different approach. This is necessary because objects in this system survive both process and machine failures. If the process in which an object is active crashes, that object will be re-activated when the next message for it arrives. The object issuing that message will be unaware that the destination object has had to be re-activated (except for a slight delay). Not surprisingly, the DOMF does not provide a notification service analogous to those provided by the other two systems. Furthermore, it is the responsibility of the re-started object to recover from the failure. Thus whiteboard objects use persistent store to hold their vital data, such as the names of the conversation participants, the object identifiers of their whiteboard objects and the object identifier of the trader object. The contents of the whiteboard display are not saved, so when a failed whiteboard object is re-started the user is presented with a blank display (a copy of the whiteboard can be obtained from one of the other participants).

A basic philosophy of Arjuna is to make failure transparent through the use of atomic actions. Changes to group membership do use atomic actions, but for reasons of efficiency we chose not to use atomic actions for the whiteboard updates, so we are primarily concerned here with the support Arjuna provides for failure detection outside atomic actions. Arjuna requires the programmer to provide status variable which can be set by a remote object to report an exception. However, ascertaining the nature of a system failure is not trivial, and would require the programmer to query the stubs. From this point of view the failure detection in Arjuna was not helpful.

4.7 Exception handling

In a distributed application, remote invocations are not guaranteed always to generate a reply. The application must deal with more failure modes than with a local procedure call. ANSAware provides the most sophisticated error handling language. An error handling routine can be associated with each remote invocation; if an exception is raised by the system (the set of possible exceptions is defined), the application can ignore it, abort the program or switch to the error handling routine. The default is to abort the program if an exception is raised. The error handling routine can set the values of the input parameters and repeat the remote invocation or abort the program (after, perhaps, printing an appropriate error message for the user).

Every remote invocation in the DOMF has a status variable associated with it. Its value can be set either by the system (in the event of the system detecting an exception) or by the application. Again a set of standard exceptions are defined.

In ISIS, the system informs the sender of the number of destinations to which the message was multicast and the number of destinations from which replies were received. If the system detects an error at a remote site (for example, if the remote object does not recognize the message type), then no reply will be received by the sender from that destination. However the sender is not told which destination has failed to reply.

Arjuna does not provide an exception handling language. This reflects the Arjuna philosophy of trying to insulate the programmer from faults as much as possible by using atomic actions to give a very clean failure semantics. Arjuna does provide an easy failure recovery mechanism, namely by aborting the atomic action. However the current version of Arjuna does not return status values to indicate the success or otherwise of a remote invocation. Attempts to communicate with a failed object will eventually time-out, although this problem may only be detected when an object attempts, and fails, to commit an atomic action involving the failed object. Thus it is advisable for all communication between objects to take place inside an atomic action so that such failures can be dealt with.

4.8 Programming support

Both ANSAware and the DOMF provide programming language independent interface definition languages. These force the programmers to agree upon a signature for
the interface. The IDL compilers for ANSAware and the DOMF produce C header files defining the data types used in the interfaces and signatures for the procedures used in the stubs. They can also produce templates for server routines including the calling parameters and return values.

In ISIS programmers must agree on the type and format of messages. No language is provided for defining these independently of the implementation. There are no tools for checking on the message types or enforcing conformance to interfaces. Arjuna uses C++ as its interface definition language.

Of the four systems only ANSAware provided a language for programming the extra constructs used in distribution, the others used a multi-function MI. The language simplified the syntax of remote invocations, and in particular the exception handling code. The preprocessor uses data from the IDL compiler to carry out extensive type checking on the use of interfaces in both client and server objects. Interface references are typed and so the system can check at compile time that interfaces are being invoked properly.

4.9 Summary

The tables 3 and 4 summarize the usefulness of the features of each platform, with regard to infrastructure and programming support. For each infrastructure we rate the feature as

- **Very useful**: the platform provided exactly what our application needed.
- **Useful**: the platform provided most of the required functionality.
- **Limited**: most of the work was done at the application level.
- **None**: all of the work was done at the application level.

5. Conclusion

We have experimented with using the groups concept in developing a distributed application. The multi-cast communication used within groups has enabled us to produce a simpler and more reliable implementation of a classic application than others who have only used point-to-point communication. Based on our experience we believe that the groups concept will be important in the development of distributed applications, especially those requiring reliability.

The fundamental difference between ISIS and the other platforms we used is that ISIS provides support for groups, while the others provide support for objects. Thus, ISIS is primarily concerned with creating groups, naming groups, maintaining group membership information and multicasting messages to group members with various delivery guarantees. The other systems are more concerned with object creation, object identity and inter-object communication. However, as we have seen, the designs of the application on the three RPC-based systems also differ significantly.

The application was ideally suited for an infrastructure which supported groups and multicast. Not surprisingly our model for the application, for example viewing each conversation as a group, mapped extremely well onto the ISIS computational model. This greatly reduced the length of time taken and the number of lines of code required to implement the application in ISIS. The other two implementations, on the DOMF and ANSAware, were comparable with each other from that point of view. Groups are being implemented in ANSAware, thus combining object and multi-point communication support in the infrastructure. We believe that support for groups should be part of every distribution infrastructure.

None of the distribution infrastructures we used provided complete support for the development and

Table 3. Infrastructure support.

<table>
<thead>
<tr>
<th>Feature</th>
<th>ANSAware</th>
<th>DOMF</th>
<th>ISIS</th>
<th>Arjuna</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groups &amp; multicast</td>
<td>None</td>
<td>None</td>
<td>Very useful</td>
<td>None</td>
</tr>
<tr>
<td>Object support</td>
<td>Useful</td>
<td>Very useful</td>
<td>Limited</td>
<td>Useful</td>
</tr>
<tr>
<td>Notification</td>
<td>Limited</td>
<td>n/a¹</td>
<td>Very useful</td>
<td>None</td>
</tr>
<tr>
<td>Failure handling</td>
<td>Limited</td>
<td>Useful</td>
<td>Very useful</td>
<td>Limited</td>
</tr>
<tr>
<td>Trading</td>
<td>Very useful</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Limited</td>
<td>Limited</td>
<td>None</td>
<td>Useful</td>
</tr>
</tbody>
</table>

¹DOMF objects are persistent and so can be re-activated by the system. It is not necessary for other objects to be notified when an object crashes or is de-activated.

Table 4. Programming support.

<table>
<thead>
<tr>
<th>Feature</th>
<th>ANSAware</th>
<th>DOMF</th>
<th>ISIS</th>
<th>Arjuna</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>Very useful</td>
<td>Very useful</td>
<td>None</td>
<td>Very useful</td>
</tr>
<tr>
<td>Type checking</td>
<td>Very useful</td>
<td>Useful</td>
<td>None</td>
<td>Useful</td>
</tr>
<tr>
<td>Exception reporting</td>
<td>Very useful</td>
<td>Useful</td>
<td>Limited</td>
<td>Limited</td>
</tr>
<tr>
<td>Threads</td>
<td>Very useful</td>
<td>Very useful</td>
<td>n/a¹</td>
<td>None</td>
</tr>
</tbody>
</table>

¹The ISIS implementation did not require threads because the system provided multicast primitives. In the ANSAware and DOMF implementations, threads were primarily used to simulate multicast.

210
A review of four distribution infrastructures

running of our whiteboard application. Considerable work needs to be done in the development of distribution infrastructures and programming tools if distributed applications are to realize the potential that advances in computer and communication technologies are offering. This paper has shown where some of that work is required.

Acknowledgments

We would like to thank Stephanie Bodoff, Nigel Derrett, and members of our group for their help in preparing this paper. The ARJUNA, ISIS, ANSA, and HP-DOMF teams provided us with support and put up with our questions in the development of the application and in the preparation of this paper—for which we are grateful.

References