HIDRA: Architecture and High Availability Support

Pablo Galdámez   Francesc D. Muñoz-Escoí   José M. Bernabéu-Aubán

Technical Report ITI-ITE-98/6

Abstract

This paper describes Hidra, a complete architecture to support the development of highly available applications in distributed systems. The components of this architecture are: first, a group membership protocol that continuously computes the valid set of nodes that are considered to belong to the system. Second, a message transport level offers reliable message passing among the nodes using the membership services. On top of this transport layer, an Object Request Broker (ORB) provides the basic object services, which include object invocation and object reference counting. Finally, replication support is provided as the approach to increase service availability in the case of domain and node failures.

This paper focuses on the object and service replication functionality provided by our architecture, where replicated objects are supported by the ORB with the help of some extra software components. Central issues are replicated object invocations which, using a kind of transactions called light weight transactions (LWT), ensure that modifications made by those invocations on the object state, are either made at every object replica or at none of them. Checkpoints are used to send the state modifications from the object replica that receives an object invocation to the rest of the object replicas. Also, a low cost and mostly asynchronous concurrency control mechanism, serializes invocations made over objects belonging to a service to maintain consistency among the replicas state.

1 Introduction

Distributed systems are a good basis to support highly available applications. In a distributed system there are multiple nodes which have independent behavior when failures arise, i.e., the failure of one of the nodes does not mean the failure of the others. So, if some support is given by the underlying system, applications can be made highly available decomposing them in components, and placing replicas of those components into independent nodes. In the rest of the paper we use the term *service* to refer to the replicable software unit, considering a highly available application to be formed by a set of replicated services. In those environments, system support has to cope with failures that affect highly available services in order to guarantee that those services continue to serve their clients as long as some of the replicas survives the failures.

Several architectures (e.g. [3, 1, 8, 15]) have been proposed to support the development of highly available applications in distributed environments. Properties such as the programming model offered for application development, the replication scheme imposed by the architecture, the computational, communication and storage cost incurred by the system to tolerate failures, the kinds of failures being tolerated and the level of concurrency allowed inside highly available applications, characterize those distinct approaches.

This paper describes Hidra, an architecture for the development of highly available, object oriented distributed applications. The programming model offered to applications is the object oriented paradigm promoted in distributed systems by the Object Management Group with the CORBA specification [20]. In Hidra, like in some of the most recent distributed environments [23, 15, 12], services are built as collections of objects which transparently access other objects that may reside in different nodes, by means of an Object Request Broker (ORB). The ORB abstracts client objects of the location of the server objects they invoke, making possible to apply those design and implementation techniques, successfully used over the last years on centralized systems to the development of distributed applications.

But the ORB itself does not suffice for the development of highly available services. The system has to offer some extra functionality to detect failures and recover the affected services from them. This support should abstract clients of services from partial systems failures, making that clients continue to access them, provided that some service replica survives the failures. Further, all this extra functionality should overhead the system as little as possible and should degrade the concurrency within services as little as possible. To achieve system availability, some approaches (e.g. [8]) use a passive replication scheme [9], where one of the service replicas is a special one, called the primary and the rest are its secondaries. A request made to the service is always processed and served by the primary which checkpoints its state to its secondaries before replying to the client, maintaining this way consistency among the replicas' state. In the case the primary fails, one of the secondaries is promoted to primary, receiving subsequent requests made to the service. Other approaches use active replication [22]. This scheme makes that every replica receives and process every single request made to the service. Group communication toolkits like [2, 5, 16, 24] are often used to preserve certain ordering among the requests received by the replicas, and relying on their deterministic behavior, all of the replicas continuously maintain the same state.

With passive replication, if the service operations are expensive in terms of computational cost, the primary replica becomes a bottleneck because it has to process every request made to the service. However, for services whose operations need few computations, or simply where load control is not an issue, its low communication cost makes it to be a satisfactory proposal . On the other hand, active replication, using a pessimistic approach, wastes computation resources given that every replica has to process every request made to the service and also, the group communication protocols they use tend to be expensive in terms of bandwidth consumption. Its principal advantage is the little time needed for failure recovery given that the required action is just to remove the failed replicas from the service group. In contrast, passive replication schemes require some additional protocols to resume the service activity after primary failures.

To overcome the major drawbacks of both approaches, Hidra allows services to be configured with an intermediate replication scheme [14] structured with a coordinator-cohort model [7]. With this strategy, each request is served by a single replica, called the coordinator of that request, which checkpoints the state modifications made by the request to the rest of the replicas, called the cohorts for that request. So, a number of primary replicas exist for a service, but just one of them act as the primary for each request. For services with expensive operations, this structure, distributes the load among the primary replicas, decreasing notably the service response time and increasing its availability. To provide even more flexibility for the range of services that may be targeted to Hidra, we allow a number of secondary replicas to be added to the service, resulting in services that may be configured as pure passive replicated ones, as pure coordinator-cohort ones or as services with any number of primary and secondary replicas.

One of the drawbacks of our replication model, that we believe is justified by the increase in the overall system performance produced by the coordinator-cohort scheme, is the requirement of a distributed concurrency control mechanism. Having that several replicas may receive different requests at the same time, it should be prevented that clients could observe an inconsistent service state. Traditionally, two-phase locking [10] has been applied as the general mechanism to preserve consistency. Even within the CORBA proposal, concurrency control mechanisms have been described for distributed systems [21], which also support nested transactions [17]. In Hidra, exploiting the object oriented model offered to applications, we propose a different, lower cost and mostly asynchronous concurrency control mechanism which borrows some of the concepts presented in [6]. In our system, when installing a service, the programmer has to provide a compatibility matrix, to express which object operations may proceed in parallel and which ones should proceed sequentially. Requests made to a given replica are allowed to proceed provided there does not exist any incompatible operation in progress within the whole service. One of the advantages of this mechanism is that concurrency control is made completely transparent to the service programmer.

Finally, the service replicas have to maintain a consistent state even when some of them fail in the middle of processing a request. CORBA services [21] propose transactions as the construction to modify the system state atomically. State modifications made by transactions are ensured to be completely made or to be completely discarded. Our approach in Hidra is to use a kind of transactions, that we call light weight transactions (LWT), attached to every request made to a replicated service. Thus, our concern is not to offer a general purpose transaction construction as the one specified by CORBA, but to offer transactions to ensure that each request made to a service is completed or its effects are completely discarded. Our approach while being less general than the CORBA one, is notably less expensive and suffices to provide an environment for the development of highly available applications.

The rest of the paper is structured as follows. Section 2 briefly outlines the Hidra architecture. Section 3 describes the Hidra ORB focusing on the support given for replicated objects. Section 4 introduces the Hidra ORB support for checkpoint objects. In section 5, we present the light weight transactions, describing the concurrency control mechanism in section 6. Section 7 explains how the architecture presented in the previous section tolerates different node and domain failures. In Section 8, we outline the further work we have initiated to complete and improve our high availability support and finally, section 9 gives a brief summary and some conclusions about this work.

2 Architecture outline

As shown in figure 1, Hidra is composed by several layers each one making use of the lower ones and providing services to the upper ones. The lowest level is a datagram transport layer that provides unreliable message transfer among all the nodes in the system.

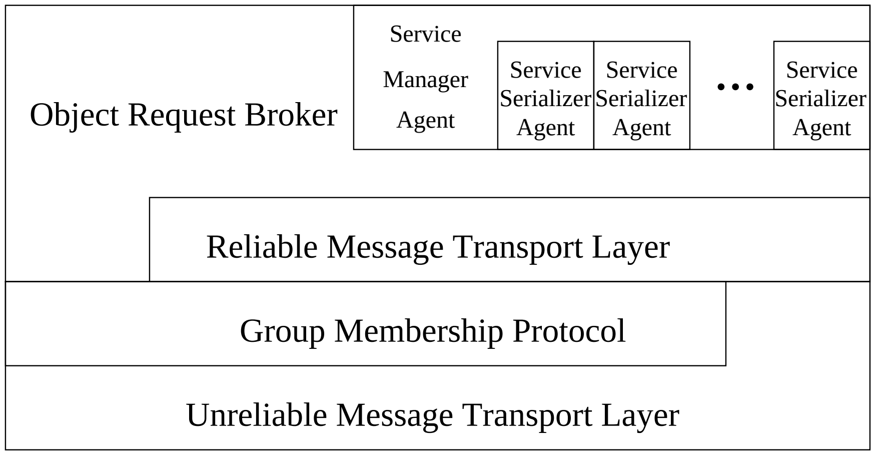

Figure 1: The Hidra code architecture

On top of this layer a group membership protocol [19] continuously monitorizes the nodes in the system to keep distributed agreement on the set of nodes that are considered to be up and running. This protocol while providing the fail-stop [] failure model used in Hidra, serves for two other basic purposes. First, when a node is computed out of the membership, every ORB in the system is notified to reconfigure itself as required. Reconfiguration is done synchronously by the ORB's of all the surviving nodes in successive steps of reconfiguration triggered by the protocol. And second, the protocol also notifies the upper transport layer of node failures to help it implement reliable message transfers. Each time the system reconfigures, a new incarnation number is set to the surviving nodes as the current incarnation number, this way, orphan messages and invalid resources that belong to failed nodes can be promptly detected.

The reliable message transport layer uses the group membership protocol to detect node failures in order to abandon communications in course with the failed nodes. It provides to the upper layers reliable message passing, where messages communications are retried until they complete or until the endpoint node is computed out of the membership. Therefore, in addition to node crashes, our failure model also tolerates low level message losses. Further, reliable messages are not guaranteed to reach their destinations at the same order as they were sent.

On top of the reliable transport layer, an ORB based on the one described in [4], provides object invocation services, and object references counting. Object invocation uses the reliable message passing provided by the transport layer, providing object invocation requests that return specific exceptions when the destination node fails. Those exceptions are sent upwards to the client object implementation that made the request or may be intercepted by the ORB itself to retry the invocation over different object replicas if the destination object was replicated. The second main function of the ORB, is to count how many nodes hold references that point to each object implementation. The goal of this reference counting is to deliver a unreferenced notification to those object implementations that do not have references pointing to them in the system. This mechanism allows to dispose the resources used by those objects when they cannot be accessed any more.

Two other distributed components complete the Hidra architecture: the *service manager* and the *service serializer*. The *service manager* (SM) provides all the administrative service operations which includes the creation and destruction of services. It also knows where each service replica is located, being involved thus, on replica registration, replica de-registration, promotion of replicas from secondary to primary, degradation of replicas from primary to secondary, participating also in the protocols run to recover from replica failures. Second, there exist a *service serializer* (SS) entity for each service installed in Hidra. This component implements a concurrency control mechanism to serialize the requests made to a service.

The service manager is itself a replicated service. To replicate it, there exists at each node a *service manager agent* (SMA) with the same interface as the SM. Some of the SMA's are primary replicas of the SM, some others are secondary replicas and some others are not SM replicas, but all of them, having the same interface, serve as the entry point for the local node operations about service management. Each SMA besides acting as the local representative of the SM at each node and possibly as one of its replicas, performs any service management operations that could be resolved locally inside a node. For instance, if two primary replicas are placed into the same node, failures that affect one of these replicas does not require the SM intervention, sufficing the local knowledge maintained by the SMA to reconfigure the service.

Figure 2 shows the state[1] stored by the Hidra service manager and the service manager agents of three Hidra nodes that maintain replicas for two services. Service 1 is configured to have at node 2 one primary replica and one secondary replica, and two additional primary replicas placed at node 3. Service 2 is built with two primaries at node 1 and two other primary replicas at node 2.

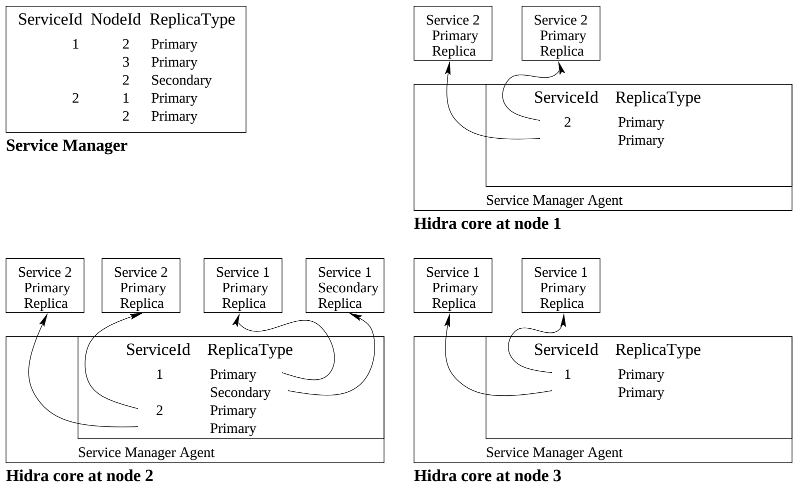

Figure 2: The service manager and the service manager agents

At each Hidra node with primary replicas for a particular service, there exists a *service serializer agent* (SSA) to manage the concurrency control needs of its local service primaries. In addition, one of the service SSA's also acts as the *service serializer*, managing the global service concurrency control requirements. The SS holds the operations compatibility matrix of the service, having this matrix replicated among a number of the SSA's. However, there is just one SS and no other one acts as a SS replica. Availability of the SS is achieved reconstructing its dynamic state about non-terminated operations running a distributed protocol among the surviving SSA's. To reduce the communication overhead required to contact the SS at each service operation, the compatibility matrix may be partitioned in sub-matrixes, placing each sub-matrix at the node where its operations are requested most frequently

3 The Object Request Broker

The Hidra ORB follows a similar design to the ORB described in [4]. Every domain in an Hidra node has access to the ORB services, where a special, trusted domain, also hosting the complete Hidra components, translates object references when inter-domain or inter-node invocations are requested. The rest of the domains, being untrusted, contain a reduced ORB structure, requiring the trusted domain intervention to access objects placed outside their address space. To increase system availability, we place the Hidra core at the operating system kernel which will be also the trusted domain. This results in the accomplishment of some other interesting properties. For instance, any operating system service such as device drivers or file systems, may access at low cost the object services and

[1]The ServiceId is a plain number that uniquely identifies any Hidra service, having the service manager assigned the ServiceId 0.

the high availability support provided by Hidra. And further, if the trusted domain fails, we have that every local domain also immediately fails, reducing to complexity that would otherwise appear if the trusted domain fails having the rest of the node still working.

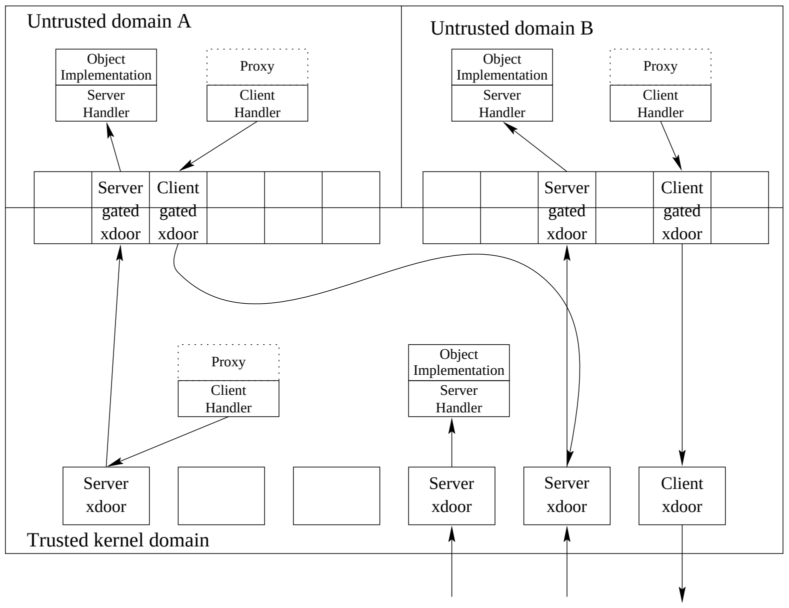

Figure 3: The Hidra Object Request Broker

The ORB structure, depicted in figure 3 shown the relationships among the handler layer and the reference layer on a Hidra node with two untrusted domains besides the trusted kernel domain. The handlers level holds client and server handlers. Each server handler is attached to one object implementation while client handlers are connected to proxy representations of the server object. Object references are C++ pointers to either the object implementation or any of its proxies, which sharing the same interface, hide the actual object location to the object reference users. Handlers are responsible for marshaling input arguments, sending them to the object implementation, waiting for a reply, and unmarshaling the output arguments. To provide object implementors with different marshaling, unmarshaling or exception handling procedures, a number of handlers[2] exist in our ORB. For instance, for objects whose marshaled representation includes a big stream of data, there exists a specialized handler providing marshaling and unmarshaling methods that avoids making unnecessary copies of the marshal stream. Other simple objects just require the object identifier to be sent to the destination, while some others run a distributed protocol to count how many object references there exist in the system pointing to the object implementation.

Each handler is connected to an xdoor from the reference layer. Server handlers are connected to server xdoors while client handlers to client xdoors. Xdoors serve as the communication endpoints for object invocation. The state of client xdoors contain the location of the server xdoor they stand for, redirecting invocations to them when invocations over client handlers are requested. Xdoors are included in the trusted domain, while in the rest of the domains, an implementation-reduced version of them, called gated-xdoors, provide the attachment from the object handlers to the xdoors residing at the kernel level. Similarly to handlers, a number of xdoor classes provide different marshaling and unmarshaling procedures. The most common xdoor type is the standard xdoor, which running a distributed protocol, counts the object references sent to remote nodes. The reference counting protocol is responsible for delivering a unreferenced notification to the object handler, when no external references point to the object implementation. This protocol built with asynchronous messages, ensures two basic properties:

- *Liveness*: Eventually, after the number of external references drops to zero, the unreferenced notification is delivered to the object handler.
- *Safety*: The unreferenced notification is only sent when the number of external references actually drops to

[2]See [4] for the complete description of the kinds of object handlers that are provided by the ORB.

zero.

To give support for object replication as the basic construction to replicate services, we make use of the extensibility mechanism provided by our ORB. We implement a new type of xdoor and a new type of handler that implement new marshaling, unmarshaling and invocation processing procedures for replicated objects. The new handler type is called the replicated object handler (ROhandler). When this handler receives the standard CORBA exception raised when the invocation destination fails, the invocation is simply retried[3]. The new type of xdoor, called the replicated object xdoor (ROxdoor), is responsible for choosing an adequate target node for each invocation it starts. The ROxdoor closely interacts with the SMA to access the current service configuration, allowing a quick redirection of invocations to living object replicas in case of failures. The close relationship among ROxdoors and the SMA also allows to block and resume all the object replica activity when processing any of the protocols that modify the service configuration.

The rest of this section further describes the replicated object support provided by our ORB, showing how replica objects can be created and how are they identified, how we guarantee the delivery of the `unreferenced` notification to every replica object, and finally, how the object invocation mechanism enables selecting any of the primary object replicas as the invocation destination incurring in very little extra cost.

3.1 Replicated Object References

In our ORB, borrowing the concept from [4], object references are implemented as pointers to C++ objects. Object references within the same domain where the object implementation resides are pointers to the actual object implementation, whereas object references at any other domain are pointers to proxy objects. However, for replicated objects, object references are always pointers to proxy objects attached to ROhandlers. For replicated object references within the same domain where a primary object replica resides, interposing the ROhandler allows these requests to be serialized, calling the service serializer before allowing the invocation to be expedited to the replica implementation.

3.2 Replicated Object Handlers

Server ROhandlers differ among other things from the other types of server handlers in that they may receive invocations from both the underlying xdoor and the proxy object to which they are also connected. When an invocation is requested from the proxy object, no argument is marshaled and the invocation is expedited to the object replica implementation, as soon as the service serializer allows the request to proceed. Both server and client ROhandlers when receiving invocations from their proxies, also start light weight transactions to increase reliability of the service requests. Those transactions are explained in section 5 while the concurrency control mechanism implemented by the serializer is commented in section 6.

The other difference between ROhandlers and the rest of the Hidra ORB handlers is the server ROhandler release operation. When releasing a non-replicated object reference placed at the same domain as the object implementation, the domain just loses the pointer to the implementation and a counter stored at the server handler is decremented. This counter stores how many local references are valid inside the server domain. When this counter drops to zero, the unreferenced notification will be sent to the implementation code as soon as the underlying server xdoor detects that its external reference counter also drops to zero. This means that the server xdoor does not need to know whether there exist or not valid object references inside the server domain. As explained in section 3.7, ROxdoors do need to know whether there exist or not valid object references inside replica domains. For this reason the release operation provided by server ROhandlers is not simply to decrement the local counter of local references but also to inform the attached server xdoor when this counter drops to zero.

For the sake of clarity, in the rest of the paper, we represent the knowledge ROxdoors have about the presence of valid object references within a domain, with a label over the attachment between ROxdoors and ROhandlers. The label values can be *client*, if the ROhandler domain contains valid object references, *primary* if the domain is a primary replica for the service and *secondary* is the domain is a secondary replica.

[3]As in Spring [11], we retry failed invocations at the handler level. Replication in Spring is made at the subcontracts level which is similar to our handlers level.

3.3 Replicated Object Xdoors

ROxdoors differ from the rest of the ORB xdoors in the way client ROxdoors point to the location of the server ROxdoor.

For non-replicated object xdoors, the client one stores the full location of the server xdoor. Specifically, the client xdoor stores the node number and the incarnation number of the server node plus the server port number and the server port version number. The server port number is the location where the server xdoor is placed at the server node, and the server port version number represents the number of times the port number has been used at the server node to store an xdoor. Invocations made from clients to servers only succeed if the receiving node finds in the incoming invocation stream the same port version number as the one stored in the indicated server port and the incarnation number of the server node is the same as the one included in the invocation stream. This one-to-one relationship solves satisfactorily invocations between clients and non-replicated servers where the only concern is to assure that both client and server agree on both the incarnation number of the nodes and on the version number of the xdoor.

For replicated objects, we find ourselves in the case that we need support to rapidly choose different destination nodes when invocation requests fail due to primary crashes. To this end, client ROxdoors do not store the destination node number, but the node chosen to serve a particular request is elected, with cooperation of the local SMA, just before invocations are sent out of a node. To fully identify which object should be the target for client invocations, client ROxdoors store the object identifier (ObjectId) of the target replicated object, together with the ServiceId of the service that contains that replicated object implementation. The ObjectId is used at the receiving node to lookup the server ROxdoor, returning an specific exception if there does not exist a primary replica for that object. Therefore, while being a little bit more expensive to find the server ROxdoor at the server node, we allow the client side, to interchangeably choose any of the nodes with primary replicas for that object at invocation time.

Server ROxdoors also differ from non-replicated object server xdoors in that they may serve as the entry point for a number of object replicas placed at the same node. So, server ROxdoors placed in the kernel, may be attached to a number of serverROhandlers. The local SMA will be responsible for choosing which of the local primary replicas will serve each invocation received by those server ROxdoors.

Finally, server ROxdoors are never released unless the replicated object reference counting protocol described in section 3.7 decides that there are no valid object references pointing to the replicated object in the whole network. For instance if a number of primary replicas exist for a particular object and just one of them holds a reference to the replicated object, the server ROxdoors attached to each replica will be active until this last object reference is released. This is required given that a valid reference placed at a replica domain, could be sent to some client domain, and this domain could choose to send the invocation to any of the primary replica domains through their server ROxdoors.

3.4 Object Identification

The cluster-wide object identifier for highly available objects is the pair (ServiceId, ObjectId). ServiceId is the service identifier to which the object belongs and ObjectId is an identifier that is unique among the objects created by this service. The ObjectId identifies the highly available object, so it identifies any of its replicas. To guarantee the uniqueness of object identifiers we build the ObjectId as the 3-tuple (NodeId, LocalOid, IncNumber[4]), where NodeId is the node identifier of the node that created the first replica of the object, IncNumber is the incarnation number of this node when creating this first object replica, and LocalOid is the counter of objects created by this node during the current cluster reconfiguration.

The ObjectId is just used as a unique identifier for replicated objects. It contains which node created the first replica of the object, but even in the case that node crashes, changing also the value of the current cluster reconfiguration, the other object replicas will continue to use the previous ObjectId as the replicated object identifier. Thus, the ObjectId as the 3-tuple (NodeId, LocalOid, IncNumber) does not necessarily mean that there exist a primary object replica at the node identified by NodeId.

[4]To guarantee the uniqueness of the ObjectId, both the LocalOid and the IncNumber should never overflow, choosing thus, as their internal representation very long integer variables. For instance, with 64-bit representations, we allow 2^64 node crashes during the system lifetime and 2^64 objects to be created at each node between two consecutive system crashes

3.5 Replicated Object Creation

A primary replica, as part of any of its operation semantics may require to create an object. As it occurs for any operation, before returning control to the operation requester, the service replica must checkpoint the modifications made by the operation to the rest of the service replicas. Given that part of the state modifications was the creation of an object, a reference to the object should be included into the checkpoint invocation, in order to allow the other service replicas to create a replica for the newly created object.

Sending the checkpoint invocation, produces the newly created object to be marshaled, and in consequence, the server ROxdoor to be created. This is the point where the ObjectId is generated. The ObjectId is generated locally by the SMA and stored together with the ServiceId into the ROxdoor state. We call this server ROxdoor, the main server ROxdoor, which is the first server ROxdoor created in the system for the replicated object. The main server ROxdoor plays a distinguished role on the replicated object reference counting protocol described in section 3.7.

When a replica domain receives a replicated object reference (for instance, as part of a checkpoint invocation), the kernel ROxdoor, a client ROhandler and the proxy are created if they did not previously exist. The ROxdoor includes in its state the ObjectId and the ServiceId. The attachment made between the ROxdoor and the ROhandler is labeled with a `client` label[5]. If the receiving domain wishes to become a replica for the object, it creates the implementation object specifying that this new implementation object is a replica. As a result a server ROhandler will substitute the previously created client ROhandler, and the SMA will be contacted to start the registration process of the new object replica. This process consists on substituting the client ROxdoor of the kernel for a server ROxdoor. This substitution is only required in the case that the ROxdoor actually was a client ROxdoor. Note that the ROxdoor could already exist as server ROxdoor if some other primary replica domain exists in the node. The second step in the replica registration process consists on re-labeling the attachment between the ROxdoor and the ROhandler with both the `primary` and *client* labels. At this point the replica domain has a valid object reference to the object while it also holds a replica of the implementation object.

3.6 Replicated Object Marshaling and Unmarshaling

As mentioned before, the first time a replicated object is marshaled from its domain, the first replica object creation process is started. This process includes creating a server ROxdoor, labeling the attachment between the object ROhandler and the ROxdoor with both the *client* and the *primary* labels, and generating the ObjectId that will be stored into the ROxdoor state. This ROxdoor is the main server ROxdoor. There is only one main server ROxdoor for each replicated object, thus the other server ROxdoors will be slightly different of this main one.

When a replicated object reference is sent to a different node, the marshal function of the ROxdoor attached to the reference is called. This function places in the marshal stream the xdoor type of ROxdoors and the ObjectId plus the ServiceId contained in the ROxdoor. Also the NodeId where the main server ROxdoor lives, is also placed into the marshal stream. The location of the main server ROxdoor is used by the replicated object reference counting protocol explained in section 3.7. The unmarshal function follows the reverse process. When the ROxdoor type is found in the incoming stream, the ROxdoor unmarshal function is invoked, which extracts the ServiceId, the ObjectId and the NodeId of the node where the main server ROxdoor resides. The local SMA is then called to lookup the ROxdoor. If the ROxdoor is not found, then it is created, assigning the received object identifier to it. Once the ROxdoor is located, a client ROhandler is created for the domain receiving the reference and the attachment made between xdoor and handler is labeled with a *client* label.

3.7 Replicated Object Reference Counting Protocol

This protocol ensures that a *unreferenced* notification will be eventually delivered to every object replica implementation of the object, some time after there is not any valid object reference pointing to the replicated object. The protocol works counting the object references as they travel around the network. To this end, ROxdoors, either server or client, have a counter, called *refcount* with the property that the addition of the refcount's of every ROxdoor for a given replicated object is, at any time, greater than or equal to the number of nodes with at least one object reference pointing to that object. Initially every ROxdoor *refcount* value is zero. The protocol works as follows:

[5]At this point, there is no knowledge of the replica intention to create a replica of the object, and the reference sent to it, is then considered as any other client object reference sent to a domain.

- Each time a replicated object reference is sent out of a node the ROxdoor whether server or client, adds one to its *refcount*.
- When a node receives a replicated object reference that results in the creation of a client ROxdoor, (there were no previous references to the object in the node and there does not exist any object replica), it sends a *INC* message to the main server ROxdoor and waits for an *ACK* message. When it receives the *ACK* it sends a *DEC* message to the domain that sent to it the reference in the first place.
- When a domain receives a replicated object reference it already had (there exists an ROxdoor whether client or server for that object), it sends a *DEC* message to the sender.
- When a ROxdoor receives a *DEC* message, it subtracts one from its *refcount*.
- When a server ROxdoor receives an *INC* message, it adds one to its *refcount* and sends an *ACK* to the sender.
- When the last client attachment between a server ROxdoor (not the main server ROxdoor) and its ROhandlers is removed, that is, when the last object reference is released, the server ROxdoor sends a *DEC* message to the main server ROxdoor.
- When the last attachment between a client ROxdoor and its ROhandlers is removed, that is, when there does not exist neither client nor servers for the replicated object in that node, a *DEC* message is sent to the main server ROxdoor.

This protocol ensures that some time after there are no valid references pointing to a replicated object, the *refcount* value of the main server ROxdoor drops to zero and there only exist server ROxdoors for that object in the network. When this occurs, and also the last object reference local to the main server ROxdoor is released (the last attachment between the main server ROxdoor and its ROhandlers labeled with *client* is removed), the main server ROxdoor contacts the local SMA to lazily send a *NOREF* message to every other server ROxdoor. Then the main server ROxdoor invokes the *unreferenced* operation of its server ROhandlers and removes itself. The same process is followed by the other server ROxdoor when they receive the *NOREF* message.

On node failures, a reference counting reconstruction protocol is run by the ORB before new object invocations are allowed to proceed. These object invocation blocking is made as the first step of system reconfiguration synchronized by our group membership protocol[19, 18]. Before blocking object invocation requests, *unreferenced* notifications are disabled.

The replicated object reference reconstruction protocol works as follows:

- Each node checks which of its ROxdoors contain as the main server ROxdoor NodeId, the identifier of the failed node. For those ROxdoors a new main server ROxdoor is elected. This election process consists of selecting as the new main server ROxdoor node, the node that containing a primary replica for the object (and thus a server ROxdoor), has as NodeId, the smallest one among the set of NodeIds which are bigger than the previous main server node, if any. Otherwise, if the previous server node had the biggest NodeId among all their replicas, then the smallest one among the remaining server nodes is chosen.
- Those references for which there does not exist a server replica, are marked as invalid and will raise an exception when used.
- Once main server ROxdoors are elected, every ROxdoor sets its *refcount* value to zero.
- Every surviving node sends a list of *INC* messages to every other node, where the list of *INC* messages sent from one node to another, contains an *INC* message for each object that having a valid object reference at the sender node, has the main server ROxdoor placed at the message destination node.
- ROxdoors receiving *INC* messages increase their *refcount*.

After the reconstruction protocol completes, *unreferenced* notifications are enabled, and then, object invocations are allowed to proceed.

3.8 Object invocation

When invoking a replicated object, the ROxdoor before sending out the invocation contacts its SMA. The SMA selects a node with a primary replica for the service and then the ServiceId and the Object Id are placed into the invocation stream. The invocation then is sent to the selected node. The receiving ORB extracts the ServiceId and the ObjectId from the incoming stream and invokes a method of the SMA to lookup the ROxdoor associated to the received object identifier. Once the ROxdoor is found, the stream will be passed to it and the ROxdoor will process it.

Our replicated object invocation scheme facilitates the task of changing from a client node the destination node where invocations are to be sent. Invocations retries due to replica failures take advantage of this simple way of changing the invocation destinations.

4 Checkpoint Support

Hidra services should use checkpoints to maintain consistency among the service replicas' state. To this end, replicas have to implement a checkpoint interface. Checkpoint messages are sent from one replica to another invoking some operation of the checkpoint interface. Our checkpointing scheme, thus requires the service programmer to define the checkpoint operations, their arguments and to implement them. Although the support we give is somehow more difficult to use than approaches like the one described in [8], it will probably be less band-width consuming and less memory and CPU time wasting. Checkpoint processing efficiency may be improved by the service programmer who can take advantage of the service semantics he implements, sending on each checkpoint invocation the minimal amount of information required to maintain state consistency among the replicas.

In order a replica to initiate a checkpoint, it has to hold a reference to a checkpoint object. Invocations made over this reference will be multicasted to every other replica for the service in the cluster. To provide this functionality, we make use again of the extensibility mechanism provided by the ORB. We introduce a new type of handler (Chkhandler) and a new type of xdoor (Chkxdoor) that basically provide multicasting services. However, we do not use group communication protocols for this purpose but we optimize message transfers among nodes when possible. For instance, if a checkpoint message has to be sent from a node, to two replicas that are placed at some other remote node, just one message is sent through the network.

Another basic difference between replicated objects and checkpoint objects, is that checkpoint objects do not expect to receive *unreferenced* notifications. Thus checkpoint object implementations will be active as long as the replica itself is active. We have that in this respect, Chkxdoors are very similar to the *simple xdoors* described in [4], that are never removed and allow clients of those objects to compute their location. Thus, to access a Chkxdoor, a remote node does not need to receive a reference to that object, but just to build it on-the-fly before accessing it.

4.1 Checkpoint Objects Creation and Identification

A replica, before becoming active for a service, has to create and register a checkpoint object. Further, all checkpoint objects defined by the replicas of a particular service must share the same interface.

Similarly to the process followed for replicated objects, creation of a checkpoint object results in the creation of a server Chkhandler attached to a proxy object which is invoked when some operation of the checkpoint object has to be invoked.

On server Chkhandler creation, the checkpoint object is registered with the SMA, which associates to it a special ObjectId. This checkpoint object identifier is just the ServiceId. The checkpoint object registration starts a protocol to add the new replica to the service as a secondary replica. Though a detailed description of this protocol is out of the scope of this paper, basically the effect it produces in the system is three fold. First, the local SMA and possibly the SM, will reflect in their state the replica addition. Second, the new replica will receive a dump of the most recent service state and third, after the protocol finishes, the replica will receive any further checkpoint invocations initiated by service replicas.

4.2 Checkpoint Multicasting

When a replica wishes to initiate a checkpoint, it invokes the reference to the checkpoint object it received when it created the checkpoint object. The server Chkhandler, instead of loopbacking this request to the checkpoint object implementation, just gives the invocation stream to the Chkxdoor. The Chkxdoor will ask the local SMA to know how many nodes in the system hold replicas for the service and upon receiving this information, proceeds to marshal the invocation stream for each destination node. To improve multicasting, each checkpoint message is sent by a different operating system thread. For synchronous checkpoint invocations, an acknowledgment is awaited for each message sent, and for asynchronous checkpoint messages control is immediately returned.

The first bytes of the invocation stream contain the Chkxdoors type and the ServiceId. Nodes receiving such invocations identified by the leading Chkxdoor type, lookup the Chkxdoor they hold for the indicated service and the invocation stream is given to it to be further processed. A Chkxdoor receiving an invocation, sends copies of the stream to the distinct server Chkhandlers that may reside at their node. Server Chkhandlers when receiving checkpoint invocations, do not immediately invoke the attached checkpoint objects but instead allocate local buffers to store the checkpoint streams. Checkpoints are stored at the Chkhandler for each light weight transaction, until every checkpoint made to complete that transaction arrives to the Chkhandler. When this happens all invocations are delivered in order to the replica implementation and the replica thus, may update its state accordingly. This checkpoint buffering facilitates to rollback[6] operations given that service replicas will not receive checkpoint invocations until the operation terminates.

4.3 Checkpoint Invocations Synchrony

Checkpoint interfaces may contain synchronous or asynchronous[7] calls. They should be used by service programmers to specify as synchronous, what we call *intention checkpoints* and the other checkpoints, called *state checkpoints* as asynchronous. Intention checkpoints require that the checkpoint requester receives an answer from each node that receives the checkpoint message before the requester returns control to the service replica domain. In a checkpoint of this type, a primary replica transmits to the other replicas its intention to modify shared state that is not volatile. So, all replicas must know about this before the actual update is made. Then, if some failure arises and the primary replica that was serving the request crashes, the replica which is elected to restart the operation can check the persistent share state to find out if the intended update was completed. Intention checkpoint acknowledgments are replied at the receiving Chkxdoors before transmitting upwards the incoming stream, but after every preceeding checkpoint has arrived to the node. This way, some gaining is achieved in the checkpoint message response time and further, checkpoint messages ordering are synchronized at each intention checkpoint message. Having every checkpoint received by a node up to an intention checkpoint allows replicas to be confident that they will receive every replica state modifications up to the intention checkpoint message in case the operation coordinator fails.

On the other hand, state checkpoints only send updates made by a operation served by a primary replica in its local state. This type of checkpoint are asynchronous[8].

In case of primary service replica failures, the primary replica elected to resume those interrupted operations that were started over the failed replica, restarts them from the last received intention checkpoint. Every replica discards from their temporal buffers, any state checkpoint received after that last intention one.

For this reason, service programmers, besides providing checkpoint operations, have to provide code to resume operations after each intention checkpoint the operation makes.

5 Light Weight Transactions

We structure highly available objects invocations as light weight transactions which are a kind of transactions that ensure that the state modification made by operations requested over a replicated serviced, are completed and their state modification are made at every service replica, or their effects completely discarded. In the rest of the section, for the sake of brevity, we use the term transactions meaning light weight transactions.

If a node or a domain receiving an invocation over a replicated object fails while processing the invocation, the operation will be automatically reinitiated over a different replica without client intervention. This is achieved by the transaction structure that we impose on each replicated object invocation. Transactions are identified during their lifetime using a small object called TID. TID's are used to reissue failed invocations over different replicas of the same object.

[6]The rollback is simply to discard the temporal buffers where the transaction checkpoint messages are stored.
[7]Asynchronous calls are specified by *oneway* operations.
[8]The sequence of asynchronous checkpoints are synchronized at once on the light weight transaction termination.

To structure invocations as transactions we benefit from the facility given by our replicated objects references to change in run-time the destination node for an invocation. We also benefit from the `unreferenced` notification to asynchronously synchronize the transaction termination. Also checkpoints are used by transactions to maintain updated the local state of any service replica, reducing the network load in case of replicas placed at the same node.

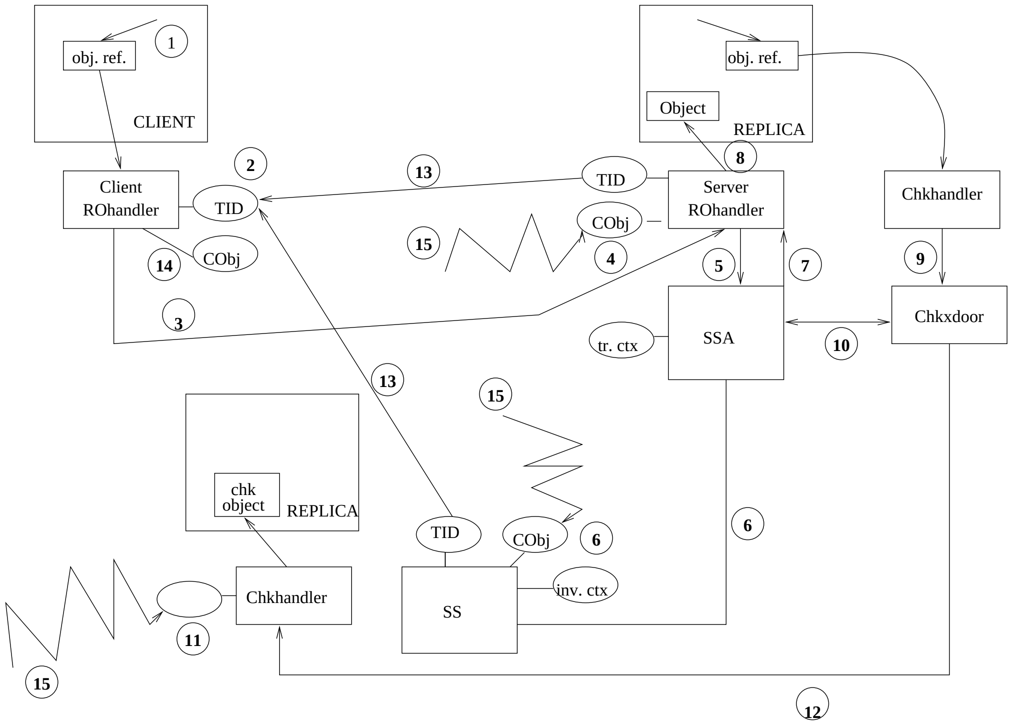

Figure 4: An example of light weight transaction

Figure 4 shows an example of a transaction initiated over a service with two primaries placed at two distinct nodes, having the client at another node.

Step 1: A client initiates a transaction when invoking a replicated object reference. *Step 2*: The ROhandler creates a TID object to identify the current transaction marshaling it as an extra argument of the request. After this, the invocation reaches the destination server ROhandler (*step 3*). *Step 4*: the ROhandler creates a small object called the *confirmation object* that is used to signal the transaction termination. *Step 5*: This server ROhandler sends a serializing request to the local SSA (The procedure followed by the SSA to serialize the request is explained in section 6) including as additional arguments the TID and the confirmation object (*Step 6*). The local SSA will block the request until the operation is allowed to proceed. The SSA besides serializing the request, internally stores the transaction context composed by the 3-tuple (TID, confirmation object, TransNumber[9]). Once the serializing request returns from the SSA (*step 7*), the replica implementation code is invoked (*step 8*) with the TID as an additional argument. The replica implementation may emit a number of checkpoint invocations until the operation terminates. Each checkpoint invocation includes the TID as an additional argument. The Chkxdoor, when receiving an invocation stream from any of its Chkhandlers (*Step 9*), contacts the SSA to know how many checkpoint messages have been delivered for the current TID (*Step 10*). If the current checkpoint message is found to be the first one, the transaction context stored at the SSA is also piggybacked into the checkpoint invocation stream. In any case the sequence number of the current checkpoint is also included into the invocation stream. This *CheckNumber* is used by the receiving replicas to order the checkpoint messages they receive, given that the transport layer does not guarantee ordered message transfers. Also, the last checkpoint sent for a light weight transaction needs to be identified, this time using a special checkpoint operation argument sent by the replica domain. It is important to detect the last checkpoint in order to terminate the transaction.

[9]The TransNumber is part of the information returned by the SS to serialize the request. How this number is generated is explained in section 6.

The *confirmation object* (CObj) is a replicated object created by the transaction coordinator and whose replicas are created by the cohort nodes (and the secondary replicas) (*step 11*) and by the serializer as soon as they receive the transaction context. The request cohorts (and the secondary replicas) receive the transaction context with the first operation checkpoint message. The serializer immediately releases this reference, and the cohort nodes will release it when they receive the last transaction checkpoint (*step 12*). The coordinator will release the reference when it replies to the client. Each cohort *signals the transaction termination* (similar to a commit request) invoking the transaction TID (*Step 13*), passing to it a reference to the confirmation object and then releasing the reference to the confirmation object they hold. The coordinator replies to the client and releases its confirmation object reference (*Step 13*) (signaling this way the transaction termination[10]). The client synchronizes the transaction termination releasing the confirmation object reference it receives (*Step 14*), as soon as it has received as many transaction termination signals as nodes with replicas there are in the network (*step 10*), excluding the operation coordinator node. We call this transaction termination synchronization the *transaction committed signal*. This release, makes that every service replica will receive on the confirmation object implementation a *unreferenced* notification (*Step 15*) which is used to remove the transaction context stored at each replica. Also, the serializer will receive that notification, that it will use to remove the associated transaction from the set of "in-progress" transactions.

Most of the messages sent around during a light weight transaction are asynchronous, allowing thus a higher degree of concurrency while processing a number of requests. The only synchronous calls are the client invocation, the serializing request (equivalent to acquiring a lock) and any required *intention checkpoints*.

6 Concurrency Control Mechanism

For each service, the concurrency control requirements are implemented by its *service serializer*. The function of the SS is to give order to the invocations received by all the objects that compose the service, taking into account incompatibilities among the service operations. Two operations are considered incompatible if they access shared state and at least one of them modifies it.

Each time a transaction arrives to a primary replica domain, the receiving server ROhandler makes a *serializing request* to the local service serializer agent (SSA) to find out which in progress transactions should terminate before allowing the current one to proceed. The SSA may be unable to locally reply to this request, redirecting in this case the request to the SS. The serializing request carries the *invocation context* together with the confirmation object and the TID. The invocation context consists of enough information to characterize the requested operation, and is used by the SSA (and possibly by the SS) to find incompatible in-progress transactions. The invocation context is composed by the method number, the ObjectId and the object class of the object being invoked. The CObj is used by the SS to be notified when this transaction finishes releasing at this moment the state stored for that transaction.

The SSA may require the SS to serialize the request. This call is always synchronous and its results are returned as soon as the SS computes which in-progress transactions are incompatible with the requested one. The arguments returned are, the TransNumber that will identify the current transaction, and the list of TransNumbers that identify all the transactions that must terminate before the current could proceed. However, the serializing request will not return from the SSA to the ROhandler that made it, until the list of predecessor transactions terminate.

The SMA's maintain the dependencies established among the current active transactions, blocking transactions until every predecessor transaction terminates.

In case of failure of the serializer, the data maintained by the SSA's permits the reconstruction of the dynamic state — the list of active transactions — of this serializer. To this end a protocol is run among the living SMA's, first elects a new SS among the SSA's and second, each SSA gives to the new SS the list of in-progress transactions.

To maintain the order provided by the serializer, the SSA has to know the current state of all transactions, either *terminated* or *active*. A transaction is considered *terminated* when the SSA has seen its last checkpoint and the invocation which carried this last checkpoint has been processed by the local replicas. Otherwise, the transaction is considered still *active*. When the SMA realizes that a transaction has terminated it removes its TransNumber from the list of preceding TransNumbers for the blocked transactions. If some of these lists becomes empty, its transaction is unblocked and it starts the operation execution.

The SSA has to maintain also a list with the TransNumbers of the terminated transactions. This list is needed to deal with the situation of an incoming transaction incompatible with some that are being terminated at the same

[10] Similar again to a transaction commit request.

time. Due to the asynchrony in the transaction termination signaling procedure, the serializer can include the TransNumbers of some transactions that have terminated or will terminate immediately in the list of predecessors for the current one. If the serialization request returns after those transactions have terminated in the local node, and the list of terminated transactions is not maintained, the just arrived transaction never will be unblocked, since part of the TransNumbers it waits for, already have been processed and forgotten.

Each TransNumber maintained by the SMA's in the list of terminated transactions is definitively released when the serializer knows that all transactions that have this TransNumber in their list of preceding TransNumbers have terminated. When this is detected, the serializer piggybacks the TransNumber to be released in the response for the following serializing request made by each SMA.

Finally, the SS may decide at any given time, to send a sub-matrix of the operations compatibility matrix to any SSA that makes serializing requests. Sub-matrixes are granted to SMA's when a big number of closely related operations are being requested by a particular SMA. This is a quite interesting property when applied to an object oriented environment like ours. It is usual to have a client domain accessing a closely related set of objects from some service, and given that the client starts invocations at one single node, the client SMA tends to select the same primary replica as the invocation destinations. Thus, if that selected replica, does not require to access the global SS, an important performance improvement is achieved. However, other clients that access operations included in the granted sub-matrix, have first to access the SS and then, the SSA where the sub-matrix resides incurring then in worse performance if those additional client accesses are frequent.

This concurrency control system provides some advantages when it is compared with other concurrency control methods, mainly with distributed locking. First, it does not produce deadlocks if the transactions only include an operation, as is the case presented here. A concurrency control system based on distributed locks might have similar characteristics if a two-phase locking protocol is used and all locks are acquired following a pre-specified order and released at the transaction's commit point. However, locks must be managed by the programmer of the highly available service, who has to know when they are needed and which type of lock must be used and where must be placed, while our support manages itself all the concurrency control problems. In our model, the application programmer only has to provide the appropriate compatibility matrix. So, our solution offers an easier programming model. Second, the dynamic information maintained by the serializer is easily recollectable in case of failures. A fault-tolerant implementation of distributed locks requires a greater amount of messages to acquire or release a lock. In our solution, moreover, the messages used to release access to a highly available object are also shared to communicate the transaction's termination. So, our concurrency control introduces a minimal overhead in the communication costs, always lower than the communication costs of a distributed lock solution.

7 Failure Recovery

This section describes how failure recovery is achieved for either node or domain crashes. Sometimes the replicated service will need to rollback accessing code the programmer has to provide while in the most cases, or either the rollback is transparent to programmers, or transactions are ensured to terminate.

To detect node failures, we use a group membership protocol. To detect replica domain failures, on replica registration, the SMA creates an object that will receive an unreferenced notification as soon as the replica crashes. To this end, the only reference to this dummy object will be given to the domain on replica registration.

Once a failure is detected, some actions are required to reconfigure the affected services and to achieve termination of the interrupted transactions. For the sake of clarity we analyze failures in terms of the actions required to recover one particular transaction that was running when the failure arose. For this interrupted transaction, we study the failure of each component and each set of components that take part in the transaction processing.

7.1 Client Failure

If only the client fails, no matter if just the domain or the whole node, no additional action is needed. The transaction will terminate, and the client failure will only be detected when the service replicas invoke the TID, but nonetheless, the transaction will be committed by the last replica in releasing its reference to the confirmation object.

On client failure thus, no extra failure recovery action is required, and we simply allow orphan transaction processing.

7.2 Cohort Failure

In either case, failures of a cohort replica domain or node crashes involving a cohort node, the required action is to update the SMA state and possibly the SM state to reflect the new service configuration. Also, if the failed cohort was the only service replica in its node, or the failure affected the whole node, the coordinator will send an invocation to the TID object with the confirmation object, to allow the client, which is waiting for a number of transaction termination signals, to receive the correct number of those message also after failures arise.

7.3 Coordinator Failure

If the coordinator fails before contacting the serializer, no extra action is required; the client node will just select the new coordinator. If the serializer received the serializing request but no cohort did receive any of the transaction's checkpoint invocations, the serializer will send a notification to all the cohorts to unblock any transaction whose serial order was set to depend on the failed transaction. Once those transactions are unblocked, the client node will be allowed to select a new coordinator.

If some cohort received a checkpoint invocation from the failed coordinator, but some of the replicas did not receive an intention checkpoint, the buffered checkpoint invocations will be discarded and the serializer will then perform the same recovery action as before.

If every cohort received at least one intention checkpoint, let be checkpoint k, the last intention checkpoint received by every replica, then every replica will be forced to process the checkpoints requests up to that checkpoint k, to upgrade their state. Any other buffered checkpoint message is safely discarded. After this is made, the client node will be allowed select a new request coordinator and the new coordinator replica will resume the operation execution from the point identified by the last intention checkpoint.

7.4 Serializer Failure

The static serializer state is replicated among a number of the SSA's, but its dynamic state is not. Thus, before allowing any further transaction processing on the service, the dynamic serializer state must be reconstructed and one of the SSA's placed at the same node as one of the primary replicas will be elected as the new serializer for the service.

To rebuild the serializer dynamic state, every SSA contacts the new serializer giving to it all the in-progress transaction processing information it holds.

Since the serializer is an object placed in the high availability ORB support, no serializer domain exists. The serializer only may crash if the node where it is running fails.

7.5 Multiple Failures

Basically, multiple failures are handled the same way as if the failures occur sequentially. The only difference is given when both the client node and the transaction coordinator fail together. In these case, the transaction has to be rolled back. To this end, first the operation is rolled back, second, the serializer will send a notification to all the surviving primary replicas to unblock any transaction whose serial order was set to depend on the failed transaction and third, every replica will release the transaction context objects.

If no intention checkpoints were received by every other replica, the rollback procedure is just to discard any checkpoint buffers held at each replica for the transaction. Otherwise, the replica rollback operation will be called. This is the only case where an expensive rollback operation could be forced for service consistency. Nevertheless this kind of rollbacks can be avoided if the programmer carefully designs the service to with operation implementations with just one intention checkpoint being the last checkpoint for the operation.

To prevent the extremely grave case that arises when the whole service crashes, the service programmer has to carefully design it, to enable recovery from state saved to secondary storage, if those failures have to be tolerated. Recall that this kind of external accesses should be done in the context of a transaction being preceded by an intention checkpoint.

8 Further Work

The design shown in this paper has to be extended to improve some aspects. Since a highly available service may request the services of other highly available services, support for nested transactions [17] is required. We are currently exploring how to extend our concurrency control mechanism to provide this support. Moreover, inter-service requests introduce a new problem that has to be solved. This problem is deadlock detection [13], that will require distributed protocols to be run among every service serializer.

At first glance, the role of the serializers should be extended to manage nested transactions. Also, each serializer should know which service originated each request made over its service, transferring this information to an external object, which could search for cycles in the path followed by multiple transactions that share some services. Once the deadlock is detected a transaction victim can be chosen to be rolled back.

We are also exploring several distinct ways to decompose the compatibility matrix in sub-matrixes to reduce as much as possible the network traffic produced by the concurrency control mechanism.

Finally, to complete our high availability support we have to describe all the service reconfiguration protocols. We have in mind protocols to create a service, remove a service, add a secondary replica to a service, promote a secondary replica to a primary one, degrade a primary replica to a secondary, remove a secondary replica from a service and to freeze a service.

9 Conclusions

Starting with an implementation of a CORBA-compliant Object Request Broker, we have presented a set of extensions to provide high availability support in object-oriented distributed programming environment. Since the ORB taken as a basis is part of the distributed operating system kernel, this support for high availability services is usable by user-level applications and also for the development of other operating system components.

There have been other attempts to provide high availability support in distributed systems, either provided by the operating system kernel itself or by user level libraries. Only a minor part of them offer support for different types of replication. Usually, when the support is given at system level, only passive replication is offered [1, 8] since the image of the primary process or replica is copied to its backup image. This is not as efficient as our solution, since the amount of data to be transferred when the state is modified may have a considerable size and can not be controlled by the application's programmer. On the other hand, if the support is given at user level, it is usually based on group communication protocols [2, 5, 16, 24] which provide message ordering to all the replicas that receive the same sequence of requests and process them concurrently. So, in that case, active replication is the immediate solution, although some toolkits offer solutions similar to ours, as the coordinator/cohort model introduced by Isis. Our alternative allows the use of secondary replicas that do not process directly any client request to the highly available object. Moreover, due to the use of the SS object and the compatibility matrix, multiple compatible requests can be served concurrently. In our solution, the number of replicas and their type are easily configurable.

Only another architecture for high availability exists offering also ORB services [15]. Its solution has been developed on top of group communication protocols, developing the ORB using group support. It also supports several types of replication models; in particular the coordinator/cohort model offered by Isis, which is very similar to ours. But our approach is different for the light weight transactions we use, the lower communication cost required by our system and the *unreferenced* facility our ORB provides for both replicated and non-replicated objects.

As we have seen, our architecture for high availability offers a set of characteristics that has not been provided by any other system or toolkit. Transactions to increase operation request reliability, the low cost incurred by the system to process them and the ability to to process multiple requests made over a service at a time are its most outstanding properties. On the other hand, our architecture offers a CORBA-like object oriented programming model for application development, where programmers just have as additional tasks, compared to non highly available application development, the design of checkpoint interfaces, the implementation of those interfaces and the specification of the operation compatibility matrix for replicated services. We believe that this extra cost is affordable, comparing it with the benefits provided by our support.

References

[1] J. E. Allchin. An architecture for reliable decentralized systems. Technical report, TR-GIT-ICS-83/23, Georgia Institute of Technology, Atlanta, Sept. 1983.

[2] Ö. Babaoğlu, R. Davoli, L. A. Giachini, and M. Baker. RELACS: A communications infrastructure for constructing reliable applications in large-scale distributed systems. Technical report, UBLCS-94-15, Dept. of Computer Science, University of Bologna, Bologna, Italy, June 1994.

[3] J. Bartlett. A nonstop kernel. In *Proceedings of the 8th ACM Symposium on Operating System Principles*, Dec. 1981.

[4] J. Bernabéu, V. Matena, and Y. Khalidi. Extending a traditional OS using object-oriented techniques. In USENIX Association, editor, *2nd Conference on Object-Oriented Technologies & Systems (COOTS), June 17–21, 1996. Toronto, Canada*, pages 53–63, Berkeley, CA, USA, June 1996. USENIX.

[5] K. P. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island, Washington*, pages 79–86, Dec. 1985.

[6] K. P. Birman, T. Joseph, and T. Raeuchle. Concurrency control in resilient objects. Technical report, TR 84-622, Dept. of Computer Science, Cornell Univ., Ithaca, NY, July 1984.

[7] K. P. Birman, T. Joseph, T. Raeuchle, and A. El-Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, 11(6):502–508, June 1985.

[8] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb. 1989.

[9] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. J. Mullender, editor, *Distributed Systems (2nd edition)*, pages 199–216. Addison-Wesley, Wokingham, England, 1993.

[10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, 1993.

[11] Graham Hamilton, Michael L. Powell, and James J. Mitchell. Subcontract: A flexible base for distributed programming. In Barbara Liskov, editor, *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 69–79, New York, NY, USA, December 1993. ACM Press.

[12] Y. A. Khalidi, J. M. Bernabéu, V. Matena, K. Shirriff, and M. Thadani. Solaris MC: A multi computer OS. In USENIX Association, editor, *Proceedings of the USENIX 1996 annual technical conference: January 22–26, 1996, San Diego, California, USA*, USENIX Conference Proceedings 1996, pages 191–203, Berkeley, CA, USA, January 1996. USENIX.

[13] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, Dec. 1987.

[14] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, Nov. 1992.

[15] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, Dept. of Computer Science, University of Zurich, Febr. 1995.

[16] C. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. Technical report, Dépt. d'Informatique, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, July 1995.

[17] J. E. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, MIT/LCS/TR-260, MIT Laboratory for Computer Science, 1981.

[18] F. D. Muñoz-Escoí, J. M. Bernabéu-Aubán, and P. Galdámez. Fault handling in distributed systems with group membership services. Technical report, ITI-ITE-98/5, Univ. Politècnica de València, Spain, September 1998.

[19] F. D. Muñoz-Escoí, Vlada Matena, J. M. Bernabéu-Aubán, and P. Galdámez. A membership protocol for multi-computer clusters. Technical report, ITI-ITE-98/4, Univ. Politècnica de València, Spain, September 1998.

[20] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, July 1995. Revision 2.0.

[21] OMG. *CORBAservices: Common Object Services Specification*. Object Management Group, Nov. 1995. Revised Edition.

[22] F. B. Schneider. Replication management using the state-machine approach. In S. J. Mullender, editor, *Distributed Systems (2nd edition)*, pages 166–197. Addison-Wesley, Wokingham, England, 1993.

[23] Iona Technologies. The orbix architecture. Technical report, Iona Technologies, Nov 1996.

[24] R. van Renesse, K. P. Birman, B. Glade, K. Guo, M. Hayden, T. M. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical report, TR95-1500, Dept. of Computer Science, Cornell University, Ithaca, NY, March 1995.