

ITRA: Inter-Tier Relationship Architecture for End-to-end QoS

Eliezer Dekel and Gera Goft

31st December 2001

IBM Haifa Research Laboratory
Haifa University
Mount Carmel
Haifa 31905
Israel
{dekel, gera}@il.ibm.com

Abstract

The importance of guaranteed end-to-end quality of service (QoS) increases as business-to-business (B2B) interactions become increasingly sophisticated, lengthy, and involve more sites. The collaboration between tiers to supply functionality can be extended to provide QoS. In this paper we focus on end-to-end availability. In multi-tier computation, support for transparent failovers is often required, even if failures occur in more than one tier at the same time. One of the obstacles in achieving a transparent failover is determining the status of outstanding operations, some of which span several tiers. Such determination of outstanding operations status upon failure requires collaboration of neighboring tiers. In this paper we present ITRA (inter-tier relationship architecture). ITRA describes mechanisms, the role of each tier with respect to its predecessor and successor tiers, programming model and inter-tier relationship protocol. Web Services following the ITRA architecture can collaborate to transparently recover from failures in multiple tiers, as well as better exploit mutual resources to provide the required availability and failover transparency aspects of an end-to-end QoS. We have exercised the architecture by mapping it to Enterprise Java Beans (EJB) and implemented a prototype.

1 Introduction

Use of Web Services over the Internet is expected to significantly increase in the near future. We envision Web Services using other Web Services, and creating contractor and sub-contractor relationships in multi-tier client/server relationships. In such environments guaranteed end-to-end quality of service (QoS) becomes very important. We argue that in order to maintain competitive service pricing Web Services' collaboration to provide a service must be extended to provide a level of service as well.

We refer to a Web Service availability as the capability to produce responses in the presence of reconfigurations. When the Web Service produces the desired responses, we say that it provides reconfiguration transparency as well. Most existing solutions neglect requests caught in the middle of a reconfiguration of a Web Service, leading to the following three problems: (1) Clients of such requests have to fix the resulted state inconsistency either at the application level or through administrator intervention. (2) Long running sessions have a higher likelihood of being caught in a failover, and restarting these sessions is in many cases unacceptable. (3) In transactional mode of operation the long detection and recovery time results in a partial denial of service.

One of the main obstacles in achieving reconfiguration transparency is the ability of a Web Service to determine the status of its outstanding operation against a Web Service in another tier. Such uncertainty can occur in the following cases:

1. A Web Service requester does not know whether or not to retry a request when it times out waiting for results. This can be easily solved with the help of the next tier if a unique sequence number is attached to each request. The next tier avoids re-executing an already executed request (identified by the attached sequence number).
2. In a replicated Web Service setting a replica servicing a request fails before returning the results; the failed over replica, receiving a retry of the same request, does not know whether or not to re-execute the request. To overcome this, the original replica can send a notification to its backup replica prior to executing a non-idempotent operation, followed by completion results. At recovery the backup replica knows whether or not to execute the non-idempotent operation (using the original sequencing).
3. In a multi-tier computation two consecutive tiers go through a failover. Partial replay for establishing an updated state may include several recent operations. A multi-tier unique and reproducible sequence numbering scheme is needed, so the same sequence numbers are attached to replayed operations. In addition, each tier has to maintain a longer history of results.

Replication is often chosen as means to achieve reconfiguration transparency. The vast majority of existing solutions replicate a Web Service state updates within its tier. For some QoS, though, strong consistency is required, resulting in synchronous replication schemes. This can be extremely inefficient if the intra-tier communication is of high latency and/or low bandwidth. This performance obstacle can be greatly reduced by integrating inter-tier state replication with careful logging of requests for replay and periodic saves of tier state snapshots in its stubs¹. We identify four questions to be addressed for reducing the performance overhead incurred by achieving reconfiguration transparency and maintaining the desired end-to-end QoS: (1) HOW to replicate - state (changes) shipping versus method shipping, message passing or shared memory, (2) WHERE to replicate - within the tier nodes or to a stub running in a client tier, (3) WHAT to replicate - any state piece or a critical subset of the state, and (4) WHEN to replicate - synchronous, asynchronous, transaction boundaries, etc.

Fault tolerance, high availability and failover transparency are widely known and researched topics for over two decades. Initial focus on fault tolerant and high performing supercomputers has switched to a wider range of solutions, starting with fault tolerant computer components, like power supply, disk, non-volatile memory, processor redundancy (SMPs), extending to tightly coupled clusters, and reaching to geographically dispersed and loosely coupled clusters. Existing research and commercial solutions in the area of multi-tier computing do not address availability and reconfiguration transparency issues uniformly. We have not seen, in existing solutions an inter-tier collaboration to achieve an end-to-end availability and reconfiguration transparency. Neither have we seen solutions addressing transparent recovery from failures in multiple tiers.

Providing end-to-end QoS for flows can be achieved by introducing enhancements to current Internet protocols. The Internet Engineering Task Force (IETF) ReSerVation Protocol (RSVP) is such a protocol [17]. This protocol is used by a host to request specific QoS from the network (routers and hosts) for particular application data streams or flows. RSVP requests will generally result in resources being reserved in each node along the data path. Applications that adaptively control routers, that support this protocol, can optimize use of resources in achieving QoS (see for example [25]). A more general approach that works on the routers level is differentiated services[16]. In this approach routers recognize packets that should receive different service based on information in the packet header.

In the intra-tier arena existing solutions replicate/share data for high availability at various levels. At the lowest operating system levels one can find shared/replicated disks [38], [39] or distributed shared/replicated memory (a comprehensive DSM survey can be found in [36]). Reliable messaging support, although, sometimes implemented at higher middleware levels, can also be considered a low level support. Examples for reliable group messaging support include an earlier MPI [56], XTP [6], MQSeries [61], Amoeba [42], ISIS [13], Delta-4 [49], PSYNC[47], Transis [5], Totem [1], RMP [62], Relacs [8], Horus [50]. More recent examples are: Ensemble [31], Maestro [57], CLUE [27], and Spinglass [14].

At higher operating system levels one can find replicated/shared file systems (Deceit [54], HA-NFS [12], GPFS [9], Pasis [63], etc.) or highly available databases [29] [7]. In a level above that, there are solu-

¹We use stub to refer to the common device used in distributed processing to abstract the remote application to the invoking application.

tions that support replication of programmable objects' via shipping of either state changing methods or state changes (CompOSE|Q [58], AQuA [20], WAFT [4], Lightweight Fault-Tolerance (LiFT)[3]). Intra-tier replication, found in various solutions, ranges from synchronous to asynchronous and asynchronous delayed (periodic) replication, and from deterministic to probabilistic[32]. Recent works (Kan [55], Easy [21]) provide a yet higher level of abstraction by masking the underlying availability/distribution support altogether, and decoupling QoS customization from application development.

Several technologies and standards have emerged to reduce the inherent difficulties of distributed computing. The Open Software Foundation (OSF) Distributed Computing Environment (DCE) [46] supports procedural programming. Object oriented programming offers several such standards, including the Object Management Group (OMG) Common Object Request Broker Architecture (CORBA) [43] [59], Microsoft's Component Object Model (COM), later on COM+ [24], and the Distributed Component Object Model (DCOM) [18], and Sun's Java 2 Enterprise Edition (J2EE) and its Enterprise Java Beans (EJB) [23] (We discuss EJB in more details in Section 6.1). The reader is referred to [48] for a comparative study of these technologies. Another example is IBM's Sun Francisco project [35] [30] that includes general infrastructure for distributed objects applications as well as domain specific components for building business applications. A recent addition that is gaining a lot of momentum is the World Wide Web Consortium (W3C), Simple Object Access Protocol (SOAP) [15] with its XML text based exchange protocol. Another technology that distinguishes itself from the rest by its focus on large-scale resource sharing is the Grid [26]. It was motivated by the computational need of scientific computing and has been evolving in several universities and national laboratories. Very little support is given in these technologies for QoS. Notable exception is the recent Fault Tolerant CORBA standard [44].

Solutions, in the inter-tier arena, for achieving transparent high availability utilize various techniques: operations logging [53], sending operations to multiple server tier nodes (Eternal [41], Fault Tolerant CORBA [44], AQuA [20]), executing operations under distributed transactional control ² [11], and caching server state at client nodes (AFS [34], Coda [37], [7]).

In this paper³ we present ITRA (inter-tier relationship architecture). ITRA describes mechanisms, the role of each tier with respect to its predecessor and successor tiers, programming model and inter-tier relationship protocol. Web Services following the ITRA architecture can collaborate to transparently recover from failures in multiple tiers, as well as better exploit mutual resources to provide the required availability and failover transparency aspects of an end-to-end QoS. ITRA addresses the handling of non-idempotent operations in such replication hybrid schemes, and it maps all availability provisioning forms, discussed earlier, into the model. The technology of ITRA is orthogonal to the popular multi-tier architectures (e.g., DCOM, CORBA) and can be naturally incorporated within those.

The rest of the paper is organized as follows: In Section 2, we describe the multi-tier computation model. We then describe the ITRA mechanisms in Section 3. ITRA's failover transparency correctness proof is given in 4. In Section 5 we discuss the applicability of ITRA to representative applications and settings. An overview of our practical experience with ITRA is presented in Section 6. In Section 7 we cover related work and in Section 8 we summarize our contribution.

2 ITRA Computation Model

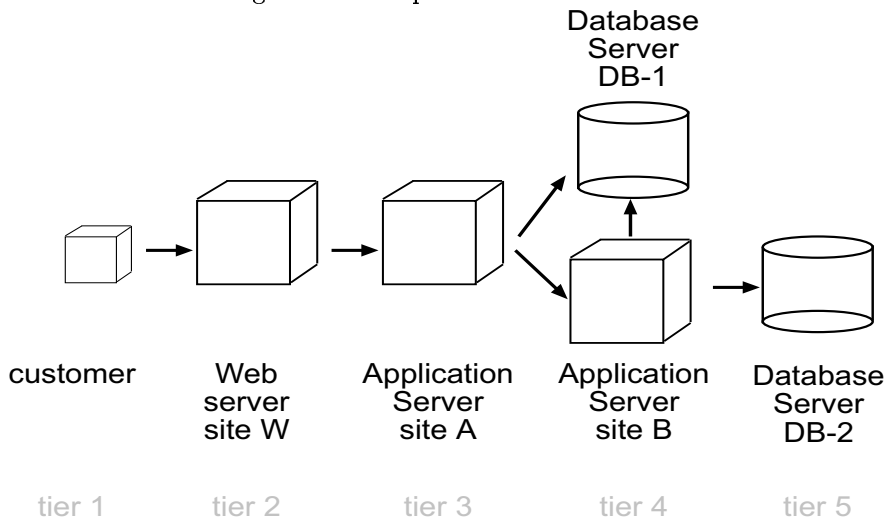
We focus on a significant and important subset of business applications/services, that are lightly or modestly compute intense. We can safely assume that each instance of an application/service can be easily computed on any node, and adding instances of computation (as a result of additional client requests) can be independently done on any node.

When discussing a computation involving a set of Web Services it is common to use a multi-tier structure (See an example in Figure 1). A tier comprises one or more related nodes that carry out certain scalability and availability QoS for the Web Service. The grouping of computers into tiers is done dynamically based on the computation flow. Tiers' separation is logical: the same node can belong to more than one tier, even

²Transactional applications are characterized by an inherent transparency (to their user) in recovering from transaction aborts, which may be triggered by failures in a server tier.

³An abbreviated version of this paper was presented in PDCS'01 [22].

Figure 1: A simple multi-tier scenario



in the same computation. In our example (Figure 1), the application server at site B (*tier 4*) applies some operation against the database server DB-1. In this situation DB-1 will appear in two roles: (*tier 4*) because of its relation to the application server in site A and (*tier 5*) because of its relation to application server in site B. Any sequence of computation is initiated at a particular node, usually referred to as the *end-client*. The end client node resides in a tier denoted as the first tier (denoted as the “customer” in our example). An end-client can be migratable (i.e., its role is assumed by another node) or distributed⁴. In the scope of this paper we only consider single node first tier.

In our model a tier stub sends an operation to a single tier node. In some of the existing solutions (e.g., [41][44]), a stub sends operations to multiple nodes. This allows for state replication via a client triggered operation shipping, and has two major advantages: 1) it can detect value failures, and 2) it imposes a lower latency than the solution of a primary propagating operations or state to backups, in settings where the inter and intra-tier communications offer similar capabilities and when the replication is synchronous. However, we have outweighed the disadvantages of such solutions: 1) value failures are statistically rare in this type of applications, as modern hardware is increasingly reliable, and the computation is not a lengthy one, and 2) typically the inter-tier communication imposes higher latency and lower throughput than the one used for intra-tier communications.

A graph where logical tiers are represented as vertices and a computation request from node in tier i to a node in tier $i + 1$ is represented by an edge is a *multi – tier computation graph*. Note that for any computation the graph is a tree. The root of the tree is the node that initiates the computation (*tier 1*). All the vertices immediately accessible from the root are in *tier 2* and so forth. Vertices that have only incoming edges are the leaves of the tree. A *multi – tier computation graph* for the example in Figure 1 is given in Figure 2.

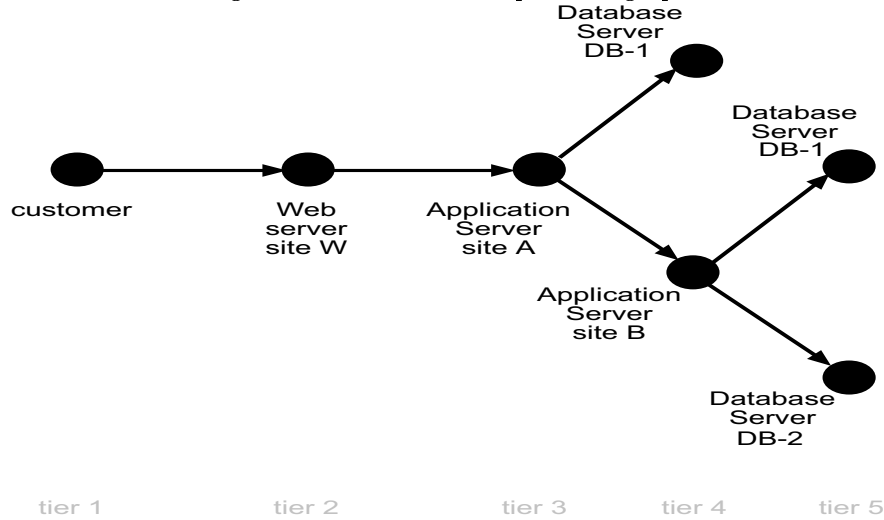
We present below a list of properties associated with an ITRA qualified computation.

Tier encapsulation: Tier’s internal structure (e.g., number of nodes, type of interconnects) is invisible outside the tier. We assume a tier has stubs representing the tier to its invoking tiers. We require a stub to be able to return state snapshots and synchronize its tier to such a presented valid state snapshot. The relationship between a tier and its stubs is part of the encapsulation. We suggest that stubs can help their tier synchronize state during failover by sending over state snapshots and re-sending logged operations. The identity of a tier’s node servicing a particular request is transparent to the invoking tier.

Resumability: We say that a Web Service has the *resumability property* if: (1) It can restart and reach a presented state (2) There is no distinction between the continuation of the execution for a copy of the

⁴In a distributed client setting sessions can logically belong to such a distributed client, rather than to a single node. The distributed client must have a mutually consistent view of the server state through any of the sessions. This can be jeopardized if the server tier partitions, and the sessions are re-connected to different sides of the partition.

Figure 2: A multi-tier computation graph



application that was started in the initial state and arrived to the given presented state and a copy that was presented with the state. To support that the Web Service has to contain a *start(state)* like method, and should be able to handle transient parts of the state adequately. Examples of potentially problematic state pieces include open sockets or file descriptors to per-node secure files or devices and open per-node database session connections.

In order to overcome failures in multiple tiers we require the server stub state to be replicated to a safe location (e.g., to another client tier node or to the client's tier stub, residing in a client's client tier). Upon recovery, the server's stub must be instantiated to a state snapshot previously taken. To achieve this each server stub has to implement a *snapshot()* method, returning its state to which it can be later instantiated, and an *start(state)* method to instantiate the stub to a presented state.

Non-idempotence: A tier executes idempotent and non-idempotent operations. An idempotent operation is one that is acting as if executed only once, even if re-executed. A non-idempotent operation must have effects that 1) can be witnessed from outside the node in which the operation is executed (e.g., sent to an external device, seen on a screen, presented to another tier), and 2) produce an inconsistent behavior if re-presented (or re-presented differently). Any operation that only results in changes of memory, registers, disk swap space, and similar state, is obviously considered idempotent. Example of a non-idempotent operation is writing into a file. We require that any non-idempotent operation acts as if it was idempotent: even if it is re-requested it must produce the results as if executed only once.

Sequential Execution: Applications and operations executed against them have a sequential order and thus during recovery operations will be executed in the same sequence as in the original execution.

In tiers consisting of multiple nodes a service can *migrate* from one tier node to another either due to a failover or due to a planned reconfiguration. With respect to server tier view a migration in a client tier is denoted as *client migration*. Excluding Byzantine behaviors the two major obstacles to overcome in achieving a transparent migration: 1) loss of an operation and 2) re-execution of a non-idempotent operation. A server tier stub has to be able to retry the operation, and a server tier has to return the operation results without re-executing it (to avoid re-execution of non-idempotent operations). A tier should be able to transparently resume any session, including an interrupted one.

Strong Affinity: We say that a client tier node has *strong affinity* to a server node, if, barring failures, the same server node continues to serve it throughout the session. We assume a strong affinity of client/server tier relationships. This is extremely important in transactional processing or in asynchronous replication. In both cases replication within a tier can occur only at certain logical points (e.g., transaction boundaries), yielding a good throughput and reasonable average latency. For this scheme to work a client tier node has to have a strong affinity to a server node at least for the duration of a session (e.g., transaction).

Recording: A tier has to maintain a log of operations and results for its client tier. The tier will return results stored in the log rather than execute them. Server logging of operations are discussed in Subsection 3.3 page 8. Suggested mechanisms for such a log are discussed in Subsection 3.5 page 9.

No re-invocation of *client synched* operations: A client tier must never re-invoke operations that where *client synched* against its server. We discuss the *client synched* operation in Subsection 3.3 page 8.

Reconstructible sequence number scheme: Each operation should be uniquely identifiable by a sequence number. Moreover, this sequence number should be reconstructible, even in the presence of failures. See Subsection 3.1 page 7 for our implementation of such a numbering scheme.

Determinism: Operations should be deterministic. The same results should be obtained when executing an operation on any of the tier nodes. Operations that are known to be non-deterministic (e.g., random number generation) should be treated as non-idempotent. Namely, the tier should take steps to keep the result of such a computation across failures.

Replication to server stubs consumes memory resources in the client tier. A client tier administrator should be able to update the maximal memory footprint of every server stub running in any of the client tier nodes by invoking an appropriate server stub method. Our architecture suggests that server stubs implement the *setMaximalMemoryFootprint(size)* method which the client's tier administration software can invoke. A server stub is responsible for updating its replication strategy accordingly (e.g., increase intra-tier replication to maintain the same failover transparency guarantees). On the other hand, a server tier, requiring a higher security level than one existing in a client tier, can decide not to replicate any state snapshots to its stubs running in that client tier. The operational effect is similar to one of decreasing the allowable memory footprint of the server stub.

3 ITRA Mechanisms

In this section we describe the extra actions to be taken during normal flow of operations across tiers. These actions include operation sequencing, client logging of requests, and server logging of results. We define the additions to each tier and its stubs residing in client tiers. We then describe the actions taken during recovery when a client tier, a server tier, or multiple tiers fail.

We introduce some notations used henceforth:

t_k - Tier k . Tiers are numbered from 1 to n according to a computation flow, where tier 1 is the initiator of a particular multi-tier computation.

o_k - Operation against t_k ; o_{k_i} - Operation i (in O_k) against t_k .

O_k - An ordered collection of operations against t_k .

$o_{k_i} < o_{k_j}$ - Means that o_{k_i} chronologically precedes o_{k_j} . Similarly greater or equal.

n_{x_k} - Node x in t_k .

s_k - Sequence number of an operation against t_k ; s_{k_i} - Sequence number of operation i against t_k .

$s_{k_i} < s_{k_j}$ - Means that s_{k_i} is smaller than s_{k_j} in lexicographic comparison. Similarly greater or equal.

ss_k - Sequence number suffix of o_k , attached by t_k 's stub; ss_{k_i} - Sequence number suffix of o_{k_i} , attached by t_k 's stub.

We say that t_k is *operation miss proof* if its state before executing any o_{k_i} is perceived by t_{k-1} as identical to the state that would have been obtained by executing a sequence of $o_{k_1} - o_{k_{i-1}}$ on a single node. In other words, such a tier never loses an operation, even in the presence of failures. We say that t_k is *duplicate operation proof* if any o_k is perceived by t_{k-1} as executed at most once. In other words, such an algorithm ensure that (non-idempotent) operations are never executed more than once. We say that t_k is *transparently recoverable* if it is both *operation miss proof* and *duplicate operation proof*. Let G be a multi-tier computation graph representing a given multi-tier computation. We define a *computation segment* in G to be a sequence of vertices representing tiers t_i to t_j , $i < j$, $j - i > 0$, such that for any $i < k \leq j$, t_{k-1} is connected to t_k . We define $state_{k_i}$ to be the state snapshot taken on a t_k node after the completion of executing o_{k_i} , and assume it contains the state snapshot of all t_{k+1} stubs and all added data structures.

3.1 Operation Sequencing

To ensure that operations are executed exactly once, t_k 's stub, running in $n_{a_{k-1}}$, has to attach a unique ID to every operation it sends to t_k . Moreover, in case of migration another instance of the same t_k 's stub, running in $n_{b_{k-1}}$ must attach the same ID for the same operation. Let the sequence number of operation $o_{(k-1)_i}$ against tier t_{k-1} be $s_{(k-1)_i}$. Assume further that $o_{(k-1)_i}$ invokes a set of operations O_k against t_k . All operations in O_k will have the same prefix $s_{(k-1)_i}$ in their sequence number. Since this is true for any instruction we can now drop the subscript i and use the more generic prefix s_{k-1} . To generate the sequence number for operation j of O_k , t_k 's stub concatenates to the received s_{k-1} a $.$ (dot) followed by ss_{k_j} . Thus, the generated sequence number s_{k_j} for operation j of O_k is $s_{k-1}.ss_{k_j}$. Below are the rules for constructing s_k :

- t_k 's stub remembers the latest received s_{k-1} , denoted by $Lpref$. $Lpref$ is initialized to zero when the stub begins its life cycle.
- t_k 's stub also remembers the latest generated ss_k , denoted by $Lsuf$. $Lsuf$ is set to zero when the stub begins its life cycle and is reset to zero whenever $Lpref$ receives a new value.
- If t_{k-1} is non-migratable⁵, then t_k 's stub will attach $s_k = 0$ value to the operations it sends to t_k . The reasoning here is that sequencing operations at t_k 's stub is only needed to uniquely identify each operation to avoid re-execution from another t_{k-1} node in case of a failover.
- If $s_{k-1} < Lpref$ then t_k 's stub returns an error (sequencing is expected to be non-decreasing); If $s_{k-1} = Lpref$, then $Lsuf = Lsuf + 1$ (another sub-operation on behalf of the same operation against t_{k-1}); otherwise $s_{k-1} > Lpref$, so $Lpref = s_{k-1}$, $Lsuf = 0$ (a new operation against t_{k-1} , so t_k 's stub can reset $Lsuf$). Concatenate $ss_k = Lsuf$ to s_{k-1} with a $.$ (dot). The result is of the form: $s_k = ss_1.ss_2...ss_{k-1}.ss_k$.

Division of responsibilities: If t_{k-1} re-invokes o_k then it must provide the same s_{k-1} , and t_k 's stub is responsible for concatenating the same suffix, ss_k to get the same s_k . If t_{k-1} migrates to another node, intending to (re)invoke o_{k_i} , then it must instantiate t_k 's stub on that node with t_k stub's state reflecting previously invoked $o_{k_{i-1}}$.

LEMMA 1: $s_{k_i} < s_{k_j}$ iff $o_{k_i} < o_{k_j}$.

The lemma implies that the operation sequencing mechanism is a monotonically increasing function of operations. It also implies that the same operation gets the same sequence in every re-invocation. Following the sequence number generation algorithm above the proof is straightforward.

3.2 Client Logging of Operations

The mechanism described in this section is optional. We present it here as a motivation and feasibility proof, as well as an example of an inter-tier collaboration to achieve a certain end-to-end QoS.

We say that o_k is *server synched* if its effects can be obtained by t_k nodes without aid from any other node, even in the presence of failures. Typically a t_k node, say n_{a_k} , executing o_k , replicates the state corresponding to o_k to (some of the) other n_{x_k} s; Each n_{x_k} acknowledges receipt of the replication; n_{a_k} then declares o_k as *server synched*. Logging an o_k in t_k 's stub helps t_k perform transparent failovers without suspending the return of o_k 's results until it becomes *server synched*. In case of failover t_k 's stub can replay o_k against a failover node in t_k in order to synchronize its state. The client node in tier t_{k-1} and server node in t_k can collaborate further by letting the server node in t_k , whenever appropriate, replicate its state to its stub in the client node in tier t_{k-1} rather than to nodes in the same tier. Thus, the server node in t_k can periodically

⁵The serving node in this tier can not fail over to another node.

send its state snapshot, corresponding to some o_{k_i} . In case of failover in tier t_k , t_k 's stub can send the snapshot state, followed by o_{k_j} , $j > i$ to help the failed over node become current. The state snapshot must also include the snapshot of the state of all stubs running in the t_k node performing the snapshot.

To avoid t_{k-1} from being cluttered t_k has to periodically notify its stub which operations are *server synched*, so that the stub can remove them from its log. We assume that the entries in the log are in the form $\langle s_k, o_k \rangle$. By Lemma 1 it is sufficient for t_k to indicate the highest s_k of the current prefix of log entries. When t_k sends a snapshot of its state, corresponding to some o_{k_i} , t_k 's stub saves the snapshot and removes the preceding operations from its log, even if some of o_{k_j} , $j \leq i$, are not *server synched*.

3.3 Server Logging of Results

Consider the case, where t_{k-1} consists of multiple nodes, and a transparent failover attempt is made from node $n_{a_{k-1}}$ to node $n_{b_{k-1}}$. Regardless of the degree of synchronization between $n_{a_{k-1}}$ and $n_{b_{k-1}}$, there is always a chance that $n_{a_{k-1}}$ will fail before notifying $n_{b_{k-1}}$ of the status of an outstanding o_{k_i} . To recover $n_{b_{k-1}}$ may need to re-execute o_{k_i} . This is unacceptable if o_{k_i} is non-idempotent. Help from t_k is needed to determine o_{k_i} 's status. If t_k has already executed o_{k_i} , it should return o_{k_i} 's results without re-executing it. For that purpose t_k maintains a log of $\langle s_k, result \rangle$ entries.

A t_k 's stub is responsible to resuming a session even if it is re-instantiated on another t_{k-1} node. The way this is achieved is beyond the requirements of ITRA, as this is precisely one of the aspects of tier encapsulation (techniques might vary with a tier implementation). One possible solution, that also catches the case where a session is lost as a result of a failover⁶, is the following: if a t_k node n_k receives an operation o_k from an unknown stub, or a new stub is created at tier t_{k-1} , n_k first queries its peers within the tier t_k about a possible past existence of a session involving that particular stub, based on the presented sequence number s_k of the first operation o_k sent by the stub to it. In case such a session existed, n_k establishes the latest available state portion, corresponding to that session.

To avoid cluttering t_k , t_{k-1} can optionally periodically notify t_k which operations will never be re-executed. We say that such operations are *client synched*. t_k can remove the results of these operations from its log. By Lemma 1, it is sufficient for t_{k-1} to indicate the highest sequence of the current prefix of log entries. In a typical implementation t_{k-1} would declare an o_k as *client synched* in one of the following three cases: (1) t_{k-1} has replicated the state corresponding to o_k 's results to (some of the) other nodes in t_{k-1} or (as part of t_{k-1} 's state snapshot) to t_{k-1} , and has received receipt acknowledgment. (2) t_{k-1} has received a *client synched* notification from t_{k-2} for o_{k-1} that triggered o_k . (3) t_{k-1} is non-migratable; in this case t_k 's stub can piggyback a *client synched* notification for o_{k_i} onto $o_{k_{i+1}}$.

A *client synched* notification is often a wave progressing from lower (client) tiers to higher (server) tiers. The proof of the mechanism correctness is given in Lemma 2 in the Correctness chapter.

3.4 Enhancements to a Tier's Stub

To maintain correct sequencing and client logging of operations each t_k 's stub maintains the following:

Fields:

Lpref - s_{k-1} attached to the latest o_k invoked against this stub

Lsuf - ss_k generate for the latest o_k by this stub

CSops - the latest *client synched* o_k

LTstate - latest t_k 's state snapshot ($state_{k_i}$ for some o_{k_i})

NotSSops - log of instructions $\langle s_k, o_k \rangle$ that are not *server synched*

MaxSize - The maximal allowed size for the stub

Migratable - This variable is *true* if t_{k-1} is migratable.

Methods:

setMaxSize() - Allows t_{k-1} set the maximal allowed size of t_k 's stub.

⁶This can happen if such a session was created on a node, but never got replicated.

setClientMigratability() - Allows t_{k-1} to indicate to a t_k stub whether it is migratable.
snapshot(kind) - Allows t_{k-1} to take snapshots (e.g., for replication) of a t_k stub.⁷

3.5 Enhancements to a Tier (Skeleton)

To maintain the tier logging of operations, t_k maintains the following information for each of its stubs:

Fields:

NotCSops - log results $\langle s_k, result \rangle$ for all o_k s that are not yet *client synched*

Migratable - a flag indicating whether or not t_{k-1} , on which the t_k stub runs, is migratable.⁸

Methods:

CSyncNeeded() - Allows t_{k+1} 's stubs request t_k to promote the latest *client synched* o_k s sequence (and thus reduce t_{k+1} 's memory footprint).

4 Failover Transparency in ITRA

4.1 Main Theorem

Consider an environment where all the assumptions mentioned in Section 2 hold.

Theorem: Let $o_{n_1} \dots o_{n_r}$ be a sequence of consecutive operations executed against t_n within a given session, and let $state_{n_j}$ be a state snapshot taken following the completion of o_{n_j} , $1 \leq j \leq r$. Let $o_{n_{j+1}} \dots o_{n_{j+w}}$ be a sequence of consecutive resulting operations executed against t_{n+1} . Let a t_{n-1} node b_{n-1} re-instantiate a t_n stub with $state_{n_j}$, and subsequently re-invoke a set of consecutive operations $o_{n_{j+1}} \dots o_{n_s}$, $s \leq r$. **Then:** (1) t_n will return the same result for any non *client synched* o_{n_i} , $j + 1 \leq i \leq s$, as if invoked for the first time, and (2) the t_n state, perceived by b_{n-1} , will correspond to o_{n_s} .

This means that relative to tier t_{n-1} , if a tier t_n has executed operations up to and including o_r within a given session, then re-instantiating an intermediate stub snapshot $state_{n_j}$, followed by a replay of any prefix of remaining operations $o_{n_j} \dots o_{n_s}$, $s \leq r$, will lead to the same perceived t_n state up until o_{n_s} ⁹, as if invoked only once. In doing that t_n will never re-execute a non-idempotent operation within the tier or a non-idempotent sub-operation against successive tiers.

Proof: a t_n encapsulates to its t_{n-1} a subgraph rooted at that t_n . Let t_m be the leaf tier with a greatest index in the computation sub-graphs resulted from the execution of any o_{n_i} , $1 \leq i \leq r$. Proof is by induction on the depth of the subgraphs $m - n$.

If $m - n = 0$ then t_n is a leaf tier. Because t_n preserves the tier encapsulation assumption and no subsequent tiers are involved, t_n will present a perceived state that corresponds to o_{n_s} , so the first claim holds. The second claim holds simply because t_n does not invoke any operations against successive tiers.

Lets assume the theorem holds for $m - n = k$, and prove for $k + 1$. From the induction, failures of t_n 's successive tiers are handled transparently. If no reconfigurations occur in t_n then t_n ignores the presented (to its stub) $state_{n_j}$ at b_{n-1} , because its state corresponds to o_{n_r} and $r \geq j$. If t_{n-1} re-invokes an o_{n_i} , $j + 1 \leq i \leq s$ then t_n does not execute it, since its state corresponds to o_{n_r} , $r \geq s$. In addition, if o_{n_i} is *client synched* t_n returns an error, otherwise t_n returns the stored results. The theorem holds. Similarly the theorem holds in the case where t_n performs a reconfiguration (e.g., a failover), establishing a state $state_{n_p}$, $s \leq p$ (and $p \leq r$), e.g., as a result of intra-tier replication. We now discuss the case where a reconfiguration occurs at t_n and a t_n node, say b_n establishes a state $state_{n_p}$, $p < s$. If $j > p$ then b_n ignores $state_{n_p}$, otherwise it establishes a state corresponding to the presented $state_{n_p}$. At the end of this step b_n maintains a

⁷Several kinds of snapshots are possible: full, incremental, partial, etc.

⁸For none-migratable t_{k-1} s maintaining *NotCSops* is unnecessary.

⁹The actual state in a given t_n node may correspond to a later operation o_{n_i} , ($s \leq i \leq r$).

state corresponding to o_{n_q} , $q = \max(j, p)$. Similarly to the above, b_n will return a result without re-executing any operation o_{n_i} , $i \leq q$. Lets consider the sequence of $o_{n_{q+1}} \dots o_{n_s}$. These operations have already been invoked against a t_n node (possibly other than b_n). Let $o_{n_{+1_u}} \dots o_{n_{+1_w}}$ be the sequence of operations invoked by t_n against one of its successive tiers t_{n+1} resulted from an appropriate subset of $o_{n_{q+1}} \dots o_{n_s}$. For proper state reconstruction up to and including o_{n_s} , we have to show that the following claims hold:

Claim 1: No non-idempotent operation is re-executed in t_n :

Because of the non-idempotence assumption t_n cannot “loose” the result if a non-idempotent operation is re-invoked with the same sequence number, and because of the sequence number reconstruct-ability assumption any non-idempotent operation must be re-invoked with the same sequence number. Hence the claim holds.

Claim 2: t_n will return the same corresponding results upon re-invocation of any o_{n_i} , $q+1 \leq i \leq s$, and t_{n+1} will be set with a state that perceptually corresponds to $o_{n_{+1_w}}$:

Any o_{n_i} , $q+1 \leq i \leq s$, that has no resulted operations against t_{n+1} tiers, will produce the same results and generated state for any re-invocation due to the determinism assumption. Lets consider all o_{n_i} , $q+1 \leq i \leq s$, that generate the set of $o_{n_{+1_u}} \dots o_{n_{+1_w}}$ against a specific t_{n+1} . We have to show that t_n preserves all “client tier” obligations with respect to t_{n+1} , which will allow us to use the induction hypothesis to show that t_{n+1} will return the same results for any non client synched replayed operation and reach the state perceptually corresponding to $o_{n_{+1_w}}$. The obligations of t_n are:

1. t_n should never require results of *client synched* operations against t_{n+1} . This is given by the client synched assumption. Note that it is possible that t_n will re-invoke *client synched* operations against t_{n+1} , but only in order to update the latter’s state (i.e., t_n will not need the results of the *client synched* replayed operations to update its state).
2. t_n has to provide the same corresponding sequence numbers when re-invoking $o_{n_{+1_u}} \dots o_{n_{+1_w}}$. Note that if a session to t_{n+1} existed when $state_{n_q}$ was taken, then t_n must have incorporated the snapshot $state_{n_{+1_{u-1}}}$. When t_n establishes $state_{n_q}$, it also presents $state_{n_{+1_{u-1}}}$ to t_{n+1} prior to invoking any following operations against it. Hence, upon invocation of the o_{n_i} , $q+1 \leq i \leq s$, that generates $o_{n_{+1_u}}$, t_{n+1} state perceptually corresponds to $state_{n_{+1_{u-1}}}$ (might be more recent). Because of the resumability, sequentiality and sequence number reconstructability assumptions, the sequential replay of all o_{n_i} , $q+1 \leq i \leq s$, will generate the same corresponding sequence numbers for $o_{n_{+1_u}} \dots o_{n_{+1_w}}$. In case a session to t_{n+1} was created chronologically later than $state_{n_q}$, then $state_{n_q}$ does not contain the session state. If such a session is “lost” (e.g., upon a failover to n_b), then t_n re-creates it using the same sequence numbers, allowing t_{n+1} to re-establish the session state (this is part of tier encapsulation discussed in Subsection 2). $o_{n_{+1_u}}$ is the first operation in the re-established session.
3. $state_{n_{+1_{u-1}}}$ corresponds to $o_{n_{+1_{u-1}}}$. $u-1$ must be in the range $1-w$ from the discussion in 2.
4. The operations against t_{n+1} , triggered from a replay of $o_{n_{q+1}} \dots o_{n_s}$ are exactly $o_{n_{+1_u}} \dots o_{n_{+1_w}}$ (the same operations and the same corresponding sequence numbers). This follows from the resumability, sequentiality and sequence number reconstructability assumptions.

We have proven that the theorem holds for a single successive tier. By the sequentiality assumption this generalizes to any number of successive tiers. This completes the proof.

Corollary 1: If t_n does not reconfigure / fail over, and all its successive tiers preserve the above assumptions, then any number of failures and re-configurations in t_n ’s successive tiers will be completely transparent to t_n , except for a possible service delay, and given that each tier eventually recovers from its failure. The proof is straightforward.

Lemma 2: The mechanism presented in Section 3.3 for server logging of results fulfills the assumption on no re-invocation of *client synched* operations (see Section 2 above).

Proof: It is sufficient to show the last step of the induction in the theorem proof above. Let o_{n+1_y} be the latest *client synched* operation against t_{n+1} (clearly, $1 \leq y \leq w$). According to the mechanism, t_n may notify t_{n+1} about *client synched*(s_{n+1_y}) in one of the following three cases: (1) t_n has replicated the state corresponding to o_{n+1_y} 's results to (some of the) other nodes in t_n or (as part of t_n 's state snapshot) to t_{n-1} and has received receipt acknowledgment¹⁰. (2) t_n has received a *client synched* notification from t_{n-1} for o_n that triggered o_{n+1_y} . (3) t_n is non-migratable; in this case t_{n+1} 's stub can piggyback a *client synched* notification for o_{n+1_y} onto $o_{n+1_{y+1}}$. In case 1 o_n will have the results in t_n regardless of whatever recoverable failure occurs in t_n or any of its preceding tiers, if they follow the sequence number reconstructability assumption. As such, o_{n+1_y} will never be re-attempted. In case 2 t_n records the latest *client synched* sequence number. If that o_n will later be re-invoked, t_n will check the latest *client synched* sequence number and return an error to t_{n-1} . Hence o_{n+1_y} will never be re-invoked. In case 3 we have to assume that the same and only t_n node does not fail (otherwise it is migratable). In such a case the t_n node contains all results of operations up to and including o_{n_n} , which the node will return without re-executing any of the operations, and hence without re-executing any resulted operations against t_{n+1} . This concludes the proof.

4.2 Configurable Parameters and Affecting Factors

ITRA enables configurable degree of failover transparency aspect of QoS. For instance, selective, lazy, or probabilistic replication may take place. Thus, a failover can be completely transparent to all clients, to some percentage of clients, or selective clients. In addition a failover can be fast for some clients and slower to other. The added value of this flexibility is that even applications that are less than 100% conforming with the ITRA model can benefit to some extent from ITRA. For instance, a tier that does not periodically take and save snapshots of its server tier stubs can still benefit from transparent failover in case of a server tier failure or even a client tier failover. The transparency vanishes only in case a failure occurs both in the server and the client tiers.

The solution ensures a multi-tier transparent recovery under a set of assumptions. Some of the assumptions may occasionally be over-restrictive, or impose significant performance degradation. A tier may not be fully enabled to ITRA. For example, it may choose not to replicate (the entire) state, not log operation results, or not be able to retrieve results of non-idempotent operations. Limited or occasionally overloaded communication bandwidth and/or high latency may deem massive replication impractical. In this section we discuss the effects of relaxing some assumptions, showing that the solution is viable for diverse settings.

If some stubs cannot run or maintain state in t_{k-1} because of security or memory footprint restriction, then t_k can still perform a transparent failover for those o_k s, that were completed prior to the failover, if replication of their corresponding state already took place within t_k . If minimal stub state size (to maintain operation sequencing) is allowed in t_{k-1} , then t_k can perform a transparent failover for the latest outstanding o_k too.

If t_{k-1} does not perform stub state snapshots either because it is not instrumented to do so or because its intra or inter (to t_{k-2}) communication overhead deems it impractical, t_k can still perform a transparent failover if t_{k-1} supports fat stubs provided that t_{k-1} does not fail.

If t_1 is migratable (e.g., when end users collaborate via other communications), then t_2 's failover transparency is only jeopardized for non-idempotent o_2 s, for which a t_1 node has not locally saved (for a checkpoint resumption) or replicated t_2 's stub corresponding state snapshot and the non-idempotent operation results to its other peer(s). Similarly, if t_k cannot obtain results of a non-idempotent o_k , it can still perform a transparent failover if it has completed replication of state that corresponds to that o_k .

t_k may perform a per-session probabilistic state replication (where the state is replicated after a o_k with a given probability), state replication for critical or some percentage of sessions, state replication at some logical computation points (e.g., at transaction boundaries or a checkpoint). t_k can also decide whether to replicate state within the tier or back to its stub. Replication can be achieved either by copying the entire state, portions of the state, log of changes, etc.

Additional properties that should be taken into consideration are the size of tier's state, the size of serialized operations, the size of the associated results and the maximal required recovery time. The above assumptions

¹⁰In such a case the corresponding state is *server synched* either within t_n , or as part of a corresponding *server synched* in t_{n-1} .

and properties determine 1) how to replicate, 2) where to replicate, 3) what to replicate, and 4) when to replicate. This demonstrates the flexibility of the architecture and its applicability to a wide range of applications and settings.

5 Representative Settings and Application Types

ITRA can be adapted to work in various settings and for different types of applications. In this section we discuss the applicability of ITRA to some representative settings and application types.

Intra-tier (distributed) shared memory: We say that a shared memory supports *update atomicity*, if it supports the following property: if one node completes performing an update of the shared memory, then a failover node will see the update at failover. Typically *update atomicity* is supported by network memory cards with separate power supply, or logical shared memory build on top of a hardware support for atomic active messages. In a setting of shared memory supporting *update atomicity* t_k keeps its state in a shared memory. There is no need for t_{k-1} to support t_k 's failover, since the state is accessible from another tier node or nodes, and as soon as t_k returns results of an o_k to its stub, the stub can safely assume that o_k is *server synched*. Consequently, there is no need for t_k to send state snapshots to its stubs. If the status of communications with t_{k+1} is also kept in the shared memory, then t_{k+1} 's support for transparent failover is not needed (even if t_k was in the middle of an o_k generating o_{k+1} s), because the status of any outstanding o_{k+1} is guaranteed to be visible to the failed over node due to the *update atomicity* property. t_{k+1} may still need to log the results of the latest outstanding o_{k+1} (if the communication between t_k and t_{k+1} is unreliable).

Intra-tier shared disk: In a shared disk setting t_k has to flush the state to the shared disk. If the state is flushed before returning the results of each o_k 's to a t_k 's stub, then the path length and CPU utilization may intolerably increase. Periodic flushing (e.g., at certain logical points of the computation) may be needed. Failover transparency for some applications may be compromised. t_{k-1} 's support for t_k 's failover may not be needed, but t_{k+1} support for o_{k+1} replay is required, because normally t_k does not synchronously update the shared disk upon receiving the results of an o_{k+1} .

LAN/WAN intra and inter-tier networks: LAN intra-tier networks allow very frequent replication rate (possibly synchronous to the operations that change state) within t_k , and for some settings (e.g., SAN), the associated CPU utilization overhead is kept low. While t_k may not need t_{k-1} support for failover, it still needs t_{k+1} support of non *client synched* o_{k+1} s.

Similarly to LAN, WAN intra-tier networks require t_{k+1} support of non *client synched* o_{k+1} s, but they also impose a higher dependency on t_{k-1} for supporting t_k 's failover. The preference to periodically push state snapshots to t_k 's stub raises if the following hold: (1) the size of such snapshots is small, (2) t_{k-1} is LAN-away, and (3) the allowed t_k 's stub memory size is sufficiently large.

For disaster recovery, where full failover transparency is not normally required, WAN connected intra-tier nodes may receive periodic/asynchronous replications of state changes, and/or have the stub instantiate the state by sending recorded snapshots and replaying remaining operations. Longer recovery time in such scenarios is tolerable.

Transactions: In the simplest scheme of transactional operation there is no need to replicate the state and logged results following each transactional operation. Rather, it is sufficient to replicate the start, abort and transaction enlist-resource operations, prepare to commit accompanied with the entire sequence of operations within that transaction, and the commit operation. Transaction tier resources are responsible for replicating their state upon receiving a prepare to commit.

Upon a failover of a transaction manager node, the transaction manager on the failed over node performs the following: it establishes the association of participating nodes/recoverable resources; if the transaction is in the prepare state, it rolls the transaction forward (commit); if the transaction is already committed it moves on to the next operation. Otherwise it rolls the transaction back. Furthermore, a tier's stub must be aware of transactional operation, and abort a transaction in case it senses a timeout when invoking a transactional operation against its tier. The reason for this is that a tier does not replicate unprepared transactional

state, so in case of a failover (potentially implied by a stub timing out), the backup node does not maintain appropriate state to resume operations within that transaction.

This simplest transactional scheme of operation offers sufficient power and eliminate the need to log operations at the stub and operation results at a tier. It also allows using the globally unique transaction IDs instead of the general sequence numbers. Yet, frequently this simple transactional scheme may be over simplistic. Specifically, we identify the following cases, for which a fully transparent failover in one or more tiers requires logging (at least some of the) operations and operation results, as well as using the general sequencing scheme:

1. Long running transactions - Aborting such a transaction is unacceptable. There are two options to avoid such aborts: (1) define short sub-transactions within the long running transaction, and abort only the latest active sub-transaction. We cover this case in discussing nested transactions below. (2) Support a completely transparent failover without losing any operation. This latter case requires our mechanisms to log operations, operation results and use the ITRA sequencing scheme.
2. Nested transactions - If a transaction does not contain operations that are not within a context of a nested transaction, then to achieve failover transparency it is sufficient to replicate the boundaries of the nested transactions and the state corresponding to the prepare to commit phase (in the same fashion described above). However, if a transaction contains operations outside the context of any nested transaction, then these operations and their results need to be logged using the ITRA mechanisms.
3. Non-transactional operations - These operations cannot be rolled back, but can still appear in a transactional sequence of operations. A fully transparent failover in the presence of such operations require logging these operations and their results, as well as using our general sequencing scheme. If all non-transactional operations are described by the developer to ITRA (or otherwise known), then it is possible to sequence and log only these operations and their results. Otherwise, all operations (and their results) must be sequenced and logged.
4. Rare or no intra-tier replication - In this case transaction and sub-transaction boundaries and resulted state (at prepare to commit phase) need to be replicated between tiers instead of within a tier. Although our general sequencing scheme is not needed here (the globally unique transaction IDs are sufficient, logging of transaction boundary operations as well as their results is needed).

Streaming applications: Streaming applications, e.g., video servers, can tolerate some state loss in case of a failover (and replay few seconds or tens of seconds) because they do not typically have non-idempotent operations, provided that failures are infrequent. We assume a streaming session is served by the same server node, unless it fails, in which case another node resumes the session. If the client tier supports stubs, then the state (which is typically small, e.g., the location in the video stream) can easily be piggybacked on the stream, and stored in the stub. In such a case a failover can be fully transparent (except for small delay in stream resumption). The intra-tier communication in this case is minimal. If the client size cannot support stubs, then applications state checkpointing is typically used: the stream server node periodically replicates the stream session status within the tier. In case of a failure another node resumes the session from the checkpoint. If the streaming application does not involve intensive client interaction any intra-tier setting is sufficient.

Dispersed collaboration applications: Dispersed collaborative applications, e.g. distributed administration and monitoring, are characterized by infrequent state updates, small to medium replication state size, and relatively tolerable latency increase. Because there may be non-idempotent operations, synchronous state replication is typically needed prior to returning results to the client. Yet, practically any shared memory/disk/file system or shared nothing LAN or WAN setting is sufficient, as well as any inter-tier communication properties. Replication can equally be done within the tier or to the client tier.

Intense collaboration applications: Intensively collaborating applications, e.g. multi-party video games or online auction drivers, are characterized by very frequent state updates, possibly large replication state size, and intolerance to latency overhead. Shared memory or shared nothing with extremely fast intra-tier LANs or SANs are the only practical settings here. Portions of the state, that are not heavily used in concurrent fashion should be replicated to the server stubs. Massively concurrently updated state portions

must be replicated within the server tier. Disaster recovery is impractical but is not normally required either. High availability and failover transparency is typically compromised, so selective state replication can be done. Shared/replicated disk is impractical because there is no one-to-one correspondence between updated state portions and the blocks that are saved/replicated to disk, and there is no clear state logical currency snapshot.

Pub/sub applications: Pub/sub applications are typically structured hierarchically to publish in a filtered fashion events to large number of subscribers (up to millions). Some of the events must not be lost. We map the pub/sub model as follows:

The first tier event providers are resource managers that generate the events. They make the generated events available to event propagators, which they treat as event subscribers in a push or pull fashion. Each event subscriber tier is the event provider to all its tiers in the next tier level. If an event must not be lost, then every provider tier records the event until all its subscriber tiers (in the next tier level) notify it that the event is *server synched*. Every subscriber possibly invokes its level filters in order to determine which of its subscribers should receive the event. At every level there are multiple subscribers, which we consider as operating independently of other tiers in the same tier level. Some events may require ordered delivery. A subscriber may spray events to different nodes in each of its subscriber tiers. This is imperative in order to achieve high throughput and scalability, but it also imposes a difficulty for those events that require ordered delivery.

Inter-tier latency is not an issue here (provided that the end user gets any event it subscribes to within seconds or minutes), but bandwidth is. Intra-tier communication should be of extremely low latency, high throughput and high basic reliability to maintain the pace of filter invocation and event ordering among the tier nodes when needed. Shared memory is best suited for this type of applications, but it typically incurs a scalability limitation, thus requiring a hybrid intra-tier setting. Non-idempotent events are possible, especially if their information is relative rather than absolute. We conclude that intra-tier state replication, reflecting arrived events, is clearly preferable because of the intra-tier interconnects properties. The replication does not need to be synchronous, but not extremely delayed either, because large amount of non *server synched* events can accumulate in the client due to massive event generation and/or propagation.

Scientific applications: Scientific applications, e.g., Grid computing [26] based applications, are characterized by bursts of very long and heavy batch computation. Non-idempotent operations are not envisioned here, and latency and throughput are not an issue. State is typically checkpointed, merely to prevent long re-computations in case of failovers. When such applications are broken to parallel sub-computations then some minimal interaction is needed among the sub-computations. Therefore, any intra and inter-tier setting is sufficient. If the stub cannot accommodate the checkpointed state, then intra-tier replication is needed. Otherwise the preferable destination of the replication is determined by the inter and intra-tier communication properties. As mentioned in the Model description section, some of the applications in this class require different operations within the same session to be routed to different server tier nodes as part of their vertical growth model. As such they do not fit into the ITRA model, which assumes load balancing at the granularity of an entire session (session affinity).

Databases: A database tier needs to be enhanced to support repeatable operations by returning the results without re-executing them. Most of the databases log operations anyway, so this enhancement is natural. In a database tier the only external visibility of operations is corresponding state changes in the database. Therefore, to avoid re-executing non-idempotent or externally visible operations completed on one node, the failed over node has to be able to retrieve the results of such operations from the database.

Frequently a database is located on a disk which is shared among the database tier nodes. In such a case, if any non-idempotent operation is flushed to disk prior to returning results to a client tier, then the failed over node can simply look into the database to discern which operations have been completed. If the database tier does not have a shared disk, then it must replicate non-idempotent operations prior to returning the results of such an operation. Note that in shared nothing settings the problem of disagreement about the completion of such an operation may surface.

Most of existing database tiers operate via dedicated sessions with clients performing an operation sequence. This ensures that the assumption, that client nodes work against the same server tier node until it fails, holds. This session support needs to be enhanced to allow both client and database tier failovers.

6 Experience

In this section we review our experience with ITRA. We chose to examine how ITRA fits with a modern multi-tier component architecture. We successfully enhanced the Enterprise JavaBeans architecture with ITRA mechanisms to provide an inter-tier component architecture that supports end-to-end QoS (Subsection 6.1). We implemented a prototype to verify ITRA's feasibility (Subsection 6.2) We conclude this section with a short discussion about performance (Subsection 6.3).

6.1 Enterprise Java Beans

To demonstrate the utility of our architecture we incorporated it into the Enterprise JavaBeans (*EJB*). The EJB specification [23] defines a multi-tier component architecture that aims at simplifying the development and integration of enterprise application by allowing the developer to concentrate solely on the business logic. The specification defines Enterprise JavaBeans (henceforth *Beans*), the interface between the Beans and the EJB *container* (a middleware layer that supplies services to the Bean), and the client to Bean relationship. The interface between the server and the container is left undefined. Our ITRA version of the EJB container adheres to the EJB specification, and by extending EJB's inter-tier encapsulation, provides an implementation that supports end-to-end QoS. We also define a container-server interface that allows the container to exploit available platform services.

A container is a collection of utility classes to enable the caching of Bean instances, interposing between method invocations, maintaining transaction integrity and general management of its pool of Bean instances in a manner, transparent to the client. To enable the container to exploit the platform on one hand and for the platform to provide services to the container on the other hand we introduced two new classes *SERVO* and *EJB RM* correspondingly. The container registers an instance of *EJB RM* with *SERVO* as a resource to be managed. It also passes to it its QoS requirements. These QoS requirements can be updated at runtime. *SERVO* is expected among other things to check resource liveness, perform restart/failovers and join, merge of resource copies. With these two classes we provide server abstraction and hide the actual platform services from the container to allow the container to be platform independent.

The EJB specification [23] allow the addition of code to the Beans by the deployment tool. Code can be added to the stubs (of *EJBHome* and *EJBObject*) and to the deployed Bean itself. This is the way vendors add implementation specific code to the Bean runtime environment. Our ITRA EJB container and deployment tool provide code to replicate state in preparation for failover. State may need to be replicated to avoid a single point of failure. State replication can be done to the next tier (e. g. DBMS), previous tier (*EJBObject* stub) or to another container in the same tier. We also provide for storing the state in EJB components. Assuming transaction manager and DBMS that are XA ([60]) compliant, we use transaction manager services and DBMS services for storing and retrieving state information whenever appropriate.

We added our stub enhancements to EJB and EJB Home stubs, and tier enhancements the *EJBObject*. We extended the EJB container provided services to include QoS support by providing a container platform interface definition. We also address the Java Transaction API/Server (*JTA/JTS*), Java Database Connectivity (*JDBC*) and Java Naming and Directory Interface (*JNDI*) components of the EJB architecture. Our architecture's flexibility is verified via EJB's stateless and stateful session beans, entity beans, EJB and EJB Home, and a variety of client connections (e.g., HTTP, RMI, IIOP). Finally, we enhance the EJB deployment descriptor with a list of optional parameters to allow tunability of QoS provision. EJB developers can either use these additional parameters to improve operation efficiency or ignore them entirely.

6.2 Prototype

We experimented with ITRA by building a prototype. The prototype models the ideas suggested in this paper. In order to get valid results, the application running on the prototype closely resembles a cut-down distributed object transaction system. The prototype is based on Java 2 Platform RMI infrastructure for services such as naming and remote method invocation. However, several features that are not part of RMI/Java are implemented by the prototype and are integrated into the distributed object usage using

smart stubs on top of the RMI stubs. These features are: Transaction propagation and Monitoring, The ITRA algorithms described in the paper, Logging of transaction states, Clustered-Distributed naming.

To demonstrate the capabilities of ITRA we executed a predefined test application (distributed transaction) on a prototype comprising 3 tiers (including the client). Once a server in a given tier failed, the underlying infrastructure identified an equivalent server, brought it up to speed, and the application continued without involving the client program in the recovery. Failing arbitrary server nodes was performed manually by simply “killing” the running component. We verified that the ITRA mechanisms function correctly under various failure scenarios.

6.3 Performance

Performance is an important factor in any architecture. Naturally, QoS has a performance price. The architecture presents many degrees of freedom in providing the required QoS. Different requirements, platform settings and operational configurations have different performance impact on different types of applications. Therefore, discussing the architecture performance in general is inherently misleading. The proposed architecture provides a rich vocabulary of setups to tune-up performance for a diversity of application types, requirements and platform settings.

7 Related Work

In this section we overview some of the existing solutions for high availability in the multi-tier (or at least the client/server) arena in order to position ITRA with respect to those.

High availability in CORBA [43], a leading inter-tier architecture, was addressed by Orbix+Isis [45], Electra [40], and recently by the fault tolerant CORBA standard (henceforth FT-CORBA) [44]. FT-CORBA was influenced by several fault tolerant CORBA projects, among them: The Distributed Object-Oriented Reliable Service (DOORS) [28], the Eternal system [41], and the AQuA system [20]. In FT-CORBA replication is done within a replication domain, and can be either cold-passive (instantiate backup and replay all operations), warm-passive (maintain backup, log operations and periodically apply portions of them on the backup), or active (all replicas handle all requests independently, and coordinate so only one replica returns results). Invocation of operations from one replication domain against other replication domains is possible, thus supporting availability in multi-tier settings.

To avoid re-execution of non-idempotent operations a client, complying with the FT-CORBA standard, attaches a request service context (unique client id, retention id, and expiration time) to each operation. Thus, allowing the server tier to identify operations that were already executed (and return their results without re-execution). While this may be sufficient for most transient inter-tier disconnections, a client migration is inadequately addressed: If a client node fails, and another client node resumes its work, then the second client node has knowledge of neither the retention id nor the expiration time of operations to be retried. This can be solved by either letting a client node replicate this information to other client nodes prior to sending the operation to a server CORBA tier (inefficient if requires message passing within the client tier), or by maintaining a globally unified multi-tier operation sequencing scheme (e.g., as the one presented by ITRA). Neither of the possibilities is addressed by the FT-CORBA standard. As such, re-execution of non-idempotent operations is at risk in the FT-CORBA if client migration occurs.

In addition, FT-CORBA does not address flexible inter-tier agreements to maintain high availability - each replication domain is autonomously responsible to maintain a certain availability and reconfiguration transparency levels. This leads to interoperability limitations, that is, replicas of an objects must be hosted by infrastructure from the same vendor. In FT-CORBA there is no support for partitioned systems. Finally, FT-CORBA uses only replication to maintain high availability (e.g., it does not exploit highly available shared memory or disk infrastructures).

The highly available AS/400 cluster prototype [7] uses intra and inter-tier replication support to maintain entirely transparent failovers in the server (database) tier. Database updates are performed against the primary replica and recorded in a log. The log entries are continuously (synchronously or asynchronously)

sent to backup replicas, where they are “applied” to the local database copy. When the primary fails, one of the backups, denoted as the first backup, assumes the primary role via a three-step recovery process: 1) Applying the remaining log entries to the local database copy 2) Receiving outstanding operations from stubs running in client nodes (and if needed, executing them) 3) Opening communication channels to clients. Yet, this work does not address sub-operations against subsequent tiers, and failures of multiple tiers, so it is sufficient that the stubs record only the latest outstanding operations. In addition, this work does not address collaboration among tiers to deliver certain level of service.

The Nile [51] project is built on top of Electra to provide a control system for hundreds of wide-area network (WAN), connected commodity processors and distributed databases that exceed 100 terabytes of data. This system uses CORBA object replication. there is no notion of replication across tiers, because there is only the client and the server tiers. all replications are among server tiers nodes. The Wide-Area Fault-Tolerance (WAFT) project [4] focuses on support for fault tolerance in WAN Object-Oriented systems, realizing that less than synchronous replication is often sufficient in such environments. Use the notion of lazy active replication that ensures that a failed over replica presents the latest sub-state to its local users and possibly an obsolete state for the rest of the replicated state. The Light Weight Fault Tolerance (LiFT) [2] project presents flexibility in providing high availability and like ITRA it also allows operation logging and a later re-start. However, this project does not discuss saving of operations at the client side, nor does it address transparent recovery from failures in cases where outstanding operations may have sub-operations (in subsequent tiers). because inter-tier cooperation to achieve failover transparency is not supported, multi-tier failures or disconnections are not supported either.

Adaptive Quality of Service for Availability (AQuA) [20], is an Ensemble/Maestro based project it provide the framework for replicating objects. Replication can be active (all replicas process request) or passive (one replica processes requests and updates state of other replicas). The proteus component configures the replicas to address dynamically changing QoS needs, initially provided by the application (there is a separation between the application and QoS, but the application developer still needs to be aware to how to achieve high availability by replication. AQuA tolerates crash failures, value failures (using active replication and result voting), and timeout failures (if a result does not arrive within a given time-frame).

Enhancing CORBA with Generic QoS management is the subject of several projects. Management Architecture for Quality of Service (MAQS) project suggests a generic QoS framework to support service policy guarantees [10]. Applications can change their behavior when QoS parameters are getting worse or better during service interaction. Another approach to QoS is suggested in Cactus [33]. The building blocks in Cactus are micro-protocols. Quality Objects (QuO) is a framework for providing QoS to software applications [64]. It is used by the AQuA project [20]. QuO’s emphasis is on specification, measuring, controlling, and adapting to changes in QoS. The basic services in ComPOSE|Q [58] are remote creation, distributed snapshot and directory services. This is a QoS enabled middleware that focuses on compose-able distributed resource management . It allows for concurrent execution of multiple resource management policies in a distributed systems in a safe and correct manner. They plan to define and enforce end-to-end QoS, and discuss QoS of the application.

A Grid version of the popular MPI that supports QoS is MPICH-GQ [52]. It uses QoS mechanisms to manage contention for resources and hence improve MPI applications. The task of providing application level QoS maps to that of configuring key parameters within individual routers. It requires the MPI application to be QoS aware hand have code to set and check the QoS parameters.

J2EE application servers vendors provide support for high availability at the intra-tier level. Some provide availability by maintaining stateful objects in a shared disk or a database. Others support a shared nothing environment where replication is done via a special implementation of JNDI. Very few vendors provide for transparent failover of server nodes. Shared nothing environments range from a group of nodes that run independently and share a dispatcher that provide load balancing to the tier to a group of node that provide complete replication of state. The low end provide only for tier scalability while the high end supports transparent failover. The JNDI implementation vary accordingly. Independent JNDI trees where each node maintains its own JNDI tree and has no knowledge about the other nodes. Shared global JNDI trees where all the nodes share the same JNDI tree and utilize a centralized name server for address resolution. Shared global JNDI allows each node to maintain its own JNDI while replicating changes to the other nodes.

Support for highly available transactions is scarce. Most servers rely on database timeouts for rollback of

transactions associated with failed nodes. This can cause locking of records for a long time and as a result denial of service for some users. Unisys has an NT base cluster server Aquanta, ES5000. According to D.H. Brown Associates, Inc, report from March 26, 1999, they have implemented a highly available transaction assistant [19]. It uses MSCS (Microsoft's cluster software) failover for Oracle DB with IP takeover of the failover node. The transaction manager that they modified is BEA Tuxedo.

One of the difficulties in integrating distributed applications is the variety of solutions and the ad-hoc manner in which applications are developed. ITRA accommodates a wide range of applications types and provides a common look-and-feel both for application developers using other applications and for tier administrators deploying such applications. ITRA complements and enhances existing distributed frameworks.

8 Conclusions

We envision a significant increase in use of commercial Web Services, and collaboration of Web Services to provide a service. The importance of guaranteed end-to-end QoS is emphasized as such B2B interactions mature and become more sophisticated, lengthy, and involve more sites. We argue that in order to maintain competitive service pricing in such environments, collaboration to provide a service must be extended to support a level of service via a better utilization of mutual resources as well. One of the aspects to be addressed is availability and reconfiguration transparency. We claim that a hybrid (inter and intra-tier) replication is often needed for full failover transparency and a better global utilization of resources.

In this paper we have presented ITRA to address these aspects. ITRA describes mechanisms, the role of each tier with respect to its predecessor and successor tiers, programming model and inter-tier relationship protocol. In ITRA we focus on business applications/services. We put no restrictions on the number of replicas or the intra-tier communication properties. Each tier can have a different support for high availability, which can optionally be reconfigured at runtime. ITRA allows complementing any intra-tier replication by dynamically adjustable inter-tier replication necessary to maintain a required end-to-end QoS. Other important factors taken into consideration are security restrictions, limitations on the allowed memory footprint of a tier-stub, services with no stubs altogether, client migratability, intra and inter-tier communication QoS, maximum allowed recovery time, replicated state size, and server tier capacity.

ITRA's flexibility allows configurable degree of failover transparency aspect of QoS. This is important when complete failover transparency may not be needed or impose significant performance degradation. It is applicable to a wide range of settings, including shared memory, shared or replicated disk, LAN/WAN intra and inter-tier communications, etc. In addition a wide range of application types can benefit from ITRA: transactional or non-transactional computations, streaming applications, dispersed or intense collaboration applications, pub/sub applications, database tiers, and other computational schemes, in which a session request is always sent to to the same WEB-service tier node, unless a failure occurs.

In our architecture we define a common look and feel for all of the approaches to achieve high availability and reconfiguration transparency. This abstraction of the level of service being provided by a tier, rather than how a tier is configured at any given time is imperative to 1) alleviate the task of Web Service developers by decoupling them from any specific configuration, 2) automate the contracts of level of service, by defining common level of service formats (e.g., in XML), and, 3) simplify the task of site administrators by presenting a higher level of management abstraction.

To prove feasibility and learn from the gained experience we enhanced an EJB container to meet the ITRA requirements. We added code and methods to EJBStub and EJBObject for stateless as well as stateful beans and bean home objects, enhanced administrator APIs, and addressed JTA/JTS, JDBS and JNDI components of the EJB architecture. We enhanced the EJB deployment descriptor with a list of optional parameters to allow tunability of QoS provision.

ITRA complements and enhances existing inter-tier architectures (e.g., FT-CORBA, DCOM SOAP, EJB).

References

- [1] D. A. Agarwal. Totem: A Reliable Ordered Delivery Protocol for Interconnected Local-Area Networks, PhD thesis, University of California, Santa Barbara, CA, USA, December 1994.
- [2] Lorenzo Alvisi, Sriram Rao and Harrick M. Vin, Lightweight Fault-tolerance for Highly Cooperative Distributed Applications, University of Texas at Austin Technical Report CS-TR-97-01, 1997.
- [3] Lorenzo Alvisi and Sriram Rao and Harrick M. Vin, Low-Overhead Protocols for Fault-Tolerant File Sharing, *International Conference on Distributed Computing Systems*, pp. 452-461, 1998.
- [4] Lorenzo Alvisi and Keith Marzullo, WAFT: Support for Fault-Tolerance in Wide-Area Object Oriented Systems, *Proceedings of the Information Survivability Workshop (ISW98)* October 28-30, 1998, Orlando, Florida, pp. 5-10.
- [5] Y. Amir, D. Dolev, S. Kramer, and D. Malki, Transis: A communication sub-system for high availability. *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 76-84. IEEE Computer Society Press, 1992.
- [6] J. W. Atwood, O. Catrina, J. Fenton, and Timothy W. Strayer, Reliable Multicasting in the Xpress Transport Protocol, *Proceedings of the 21st Local Computer Networks Conference*, Minneapolis, Minn., 1996.
- [7] A. Azagury, D. Dolev, G. Goft, J. M. Marberg, J. Satran. Highly Available Cluster: A Case Study. *FTCS 1994*.
- [8] O. Babaoglu, R. Davoli, L. Giachini, and M. Baker, Relacs: A communications infrastructure for constructing reliable applications in large-scale distributed systems, Technical Report UBLCS-94-15, Laboratory for Computer Science, University of Bologna, Italy, 1994.
- [9] J. Barkes, M. R. Barrios, F. Cougard, P.G. Crumley, D. Marin, H. Reddy, T. Thitayanum, GPFS: A Parallel File System. IBM Redbook SG 24-5165-00, 1998.
- [10] C. Becker and K. Geihs, Generic QoS-Support for CORBA, *Proceedings of 5th IEEE Symposium on Computers and Communications (ISCC'2000)* Antibes/France.
- [11] Philip A. Bernstein and Eric Newcomer, *Principles of Transaction Processing*, Morgan Kaufman Series in Data Management Systems, Morgan Kaufmann Publishers, January 1997.
- [12] A.K. Bhide and E.N. Elnozahy and S.P. Morgan, A Highly Available Network File Server, *Proceedings of the USENIX Winter Conference 1991*, pp. 199-205, Jan 1991.
- [13] K. P. Birman and T. A. Joseph, Exploiting virtual synchrony in distributed systems, *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987.
- [14] Kenneth Birman, Mark Hayden, Ozgur Ozkasap, Zhen Xiao, Mihai Budiu and Yaron Minsky, Bimodal Multicast. *ACM Transactions on Computer Systems*, (May, 1999), Volume 17, No. 2.
- [15] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, Dave Winer, *Simple Object Access Protocol (SOAP) 1.1*, W3C, May 2000.
- [16] R. Braden, D. Clark, and S. Shenker, *RFC 1633: Integrated services in the Internet architecture: an overview*, Internet RFC 1633, July 1994.
- [17] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin, *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification*, RFC 2205, September 1997, IETF, Proposed Standard.
- [18] Nat Brown and Charlie Kindel, *Distributed Component Object Model Protocol DCOM/ 1.0*, Microsoft Corporation, January 1998. Internet RFC draft 2.

- [19] D.H. Brown Associates, *Unisys Takes Different Tack on High Availability for NT*, March 1999.
- [20] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz, AQuA: An Adaptive Architecture That Provides Dependable Distributed Objects, *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS'98)*, West Lafayette, Indiana, USA, October 20-23, 1998, pp. 245-253.
- [21] E. Dekel and G. Goft, EASY Customization for Flexible QoS. In preparation.
- [22] E. Dekel, and G. Goft, ITRA: Inter-Tier Relationship Architecture for End-to-End QoS, *Proceedings of the thirteenth IASTED Symposium on Parallel, Distributed Computing and Systems*, August 2001.
- [23] Linda G. DeMichiel, L. Umit Yalcinalp, & Sanjeev Krishnan, Enterprise JavaBeans Specification, Version 2.0, Proposed Final Draft, Sun Microsystems, October 18, 1999.
- [24] Guy Eddon and Henry Eddon, *Inside COM+ Base Services*, Microsoft Press, 1999.
- [25] I. Foster, A. Roy, and V. Sander, A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation, *8th International Workshop on Quality of Service*, June 2000 pp. 181-188.
- [26] I. Foster, C. Kesselman, and S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations, (to be published in Intl. J. Supercomputer Applications, 2001).
- [27] G. Goft and E. Lotem. CLUE - The AS/400 Cluster Engine: A Case Study, *Proceedings of the 1999 ICPP Workshops*, September 1999.
- [28] A. Gokhale, B. Natarajan, D. C. Schmidt, and S. Yajnik, DOORS: Towards High-performance Fault-Tolerant CORBA, *Proc. of the 2nd International Symposium on Distributed Objects and Applications(DOAO0)*, OMG, Antwerp, Belgium, September 2000.
- [29] Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, September 1992.
- [30] Steven L. Halter and Steven J. Munroe, *Enterprise Java Performance*, Prentice Hall 2001
- [31] Mark Hayden, The Ensemble System Cornell University Technical Report, TR98-1662, January 1998. or M. G. Hayden, The Ensemble System, Ph. D. thesis, Cornell University, 1998.
- [32] Abdelsalam A. Helal, Abdelsalam A. Heddaya, and Bharat B. Bhargava, *Replication Techniques in Distributed Systems* MCC, Kluwer Academic Publishers, August 1996.
- [33] Matti A. Hiltunen, Richard D. Schlichting, and Gary Wong, Implementing Integrated Fine-Grain Customizable QoS using Cactus, *The 29th Annual International Symposium on Fault-Tolerant Computing*, June 1999, pp. 59-60.
- [34] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham and Michael J. West, Scale and performance in a distributed file system, *ACM Transactions on Computer Systems* Volume 6, No. 1 (Feb. 1988), pp. 51 - 81.
- [35] Verlyn Johnson, The San Francisco project: business process components and infrastructure, *ACM Computing Surveys*, Volume 32 , No. 1es (Mar. 2000), pp. 25 - 29.
- [36] A. Judge, P.A. Nixon, V.J. Cahill, B. Tangney, S. Weber. Overview of distributed shared memory, Technical report of the Distributed Systems Group at Trinity College in Dublin, Ireland. October 1998.
- [37] J. J. Kistler and M. Satyanarayanan, Disconnected Operation in Coda File System, *ACM Transactions on Computing Systems*, 10:1, February 1992, pp 3-25.
- [38] Yoshimichi Kosuge and Christoph Krafft, RSC T Group Services: Programming Cluster Applications, IBM Red Book, April 2000, International Technical Support Organization.

- [39] Frank Kyne, Luiz Fadel, Michael Ferguson, Rafael Garcia, Keith George, Masayuki Kojima, Alan Murphy, and Henrik Thorsen, *OS/390 Parallel Sysplex Configuration, Volume 1: Overview*, SG24-5637-00, IBM Red Book, September 2000, International Technical Support Organization.
- [40] S. Maffei, Run- Time Support for Object- Oriented Distributed Programming, Ph. D. thesis, University of Zurich, 1995.
- [41] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. Tewksbury and V. Kalogeraki, The Eternal System: An Architecture for Enterprise Applications, *International Enterprise Distributed Object Computing Conference*, University of Mannheim, Germany (September 1999), pp. 214-222.
- [42] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and J.M. van Staveren, Amoeba - a distributed system for the 1990s, *Computer*, 23(5), 1990.
- [43] Object Management Group, The Common Object Request Broker: Architecture and Specification, 2.0 Edition, July 1995.
- [44] Object Management Group, Fault Tolerant CORBA Specification 1.0, OMG Document ptc/00-04-04 edition, April 2000.
- [45] Iona Orbix product family <http://www.ionac.com/products/orbhome.htm> .
- [46] Open Software Foundation, OSF DCE Introduction to OSF, DCE Release 1.1, Prentice Hall, August 1995.
- [47] L. L. Peterson, N. C. Bucholz, and R. D. Schlichting, Preserving and using context information in interprocess communication, *ACM Transactions on Computer Systems*, 7(3):217-246, 1989.
- [48] F. Plasil and M. Stal, An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM, *Software Concepts & Tools*, vol. 19, no. 1, Springer 1998.
- [49] D. Powell, editor, Delta-4: A Generic Architecture for Dependable Distributed Computing, ESPRIT Research Reports, Springer-Verlag, 1991. Project 818/2252.
- [50] Robbert van Renesse, Kenneth P. Birman and Silvano Maffei, Horus, a flexible Group Communication System, *Communications of the ACM*, April 1996.
- [51] A. Ricciardi, M. Ogg, and F. Previato, Experience with distributed replicated objects: The Nile project, *Theory and Practice of Object Systems*, 4(2):107-117, 1998.
- [52] A. Roy, I. Foster, W. Gropp, N. Karonis, V. Sander, and B. Toonen, MPICH-GQ: Quality-of-Service for Message Passing Programs, *Proceedings of SC2000*, November, 2000.
- [53] Michel Ruffin, A Survey of Logging Uses, Broadcast Technical Report 36, Esprit Basic Research Project 6360, February 1995.
- [54] A. Siegel, K. P. Birman, K. Marzullo, Deceit: A Flexible Distributed File System, *USENIX Summer 1990*.
- [55] Jerry James and Ambuj K Singh, Design of the Kan Distributed Object System, *Concurrency: Practice & Experience*, 12(8), July 2000.
- [56] Marc Snir, Steve Otto, Steve Huss-Lederman, David Walker, and Jack Dongarra, *MPI-The Complete Reference*, Volume 1 - The MPI-1 Core, 2nd edition, The MIT Press, 1998.
- [57] A. Vaysburd and K. P. Birman, The Maestro Approach to Building Reliable Interoperable Distributed Applications with Multiple Execution Styles, *Theory and Practice of Object Systems*, vol. 4, no. 2, 1998.
- [58] Nalini Venkatasubramanian, " ComPOSE|Q - A QoS-enabled Customizable Middleware Framework for Distributed Computing", Distributed Middleware Workshop, *Proceedings of the IEEE Intl. Conference on Distributed Computing Systems (ICDCS '99)*, June 1999.

- [59] Steve Vinoski, CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments, *IEEE Communications Magazine*, Vol. 14, No. 2, February, 1997.
- [60] Andreas Vogel and Madhavan Rangarao, *Programming with Enterprise JavaBeans, JTS, and OTS: Building Distributed Transactions with Java and C++*, John Wiley & Sons, April 1999.
- [61] Dieter Wackerow, David Armitag and Tony Skinner, MQSeries Version 5.1 Administration and Programming Examples, SG24-5849-00, IBM Red Book, December 1999, International Technical Support Organization.
- [62] R Brian Whetten, Todd Montgomery, and Simon Kaplan. A high performance totally ordered multicast protocol. *Theory and Practice in Distributed Systems*, volume LCNS 938. Springer Verlag, 1994.
- [63] Jay J. Wylie, Michael W. Bigrigg, John D. Strunk, Gregory R. Ganger, Han Kiliccote, Pradeep K. Khosla. Survivable Information Storage Systems, *IEEE Computer*, August 2000.
- [64] J. Zinky, D. Bakken, R. Schantz. Architectural Support for Quality of Service for CORBA Objects, *Theory and Practice of Object Systems*, April, 1997.