Proceedings 2nd International Workshop on Configurable Distributed Systems, Pittsburgh PA, March 21—23, 1994, pp. 36—45

Application-Transparent Fault Tolerance in Distributed Systems

Thomas Becker

Computer Science Department
University of Kaiserslautern, P.O. Box 3049, D-67653 Kaiserslautern, Germany
email: tbecker@informatik.uni-kl.de

Abstract

We present a new software architecture in which all concepts necessary to achieve fault tolerance can be added to an application automatically without any source code changes. As a case study, we consider the problem of providing a reliable service despite node failures by executing a group of replicated servers. Replica creation and management as well as failure detection and recovery are performed automatically by a separate fault tolerance layer (ft-layer) which is inserted between the server application and the operating system kernel. The layer is invisible for the application since it provides the same functional interface as the operating system kernel, thus making the fault tolerance property of the service completely transparent for the application. A major advantage of our architecture is that the layer encapsulates both fault tolerance mechanisms and policies. This allows for maximum flexibility in the choice of appropriate methods for fault tolerance without any changes in the application code.

1. Motivation

Due to the steadily increasing complexity of application programs more and more sophisticated concepts are required to make these programs resistent against system component failures. During the past years several methods for achieving this fault tolerance have been developed and meanwhile are well understood. Basic concepts which are necessary for reliable applications have been identified and thoroughly studied (process surveillance [Bec91], membership information distribution [Cri91], checkpointing and recovery [KoT87], reliable and order-preserving multicast protocols [BeG93], various replication mechanisms [Pow91], etc.).

However, up to now little has been done to support a non-expert application programmer in building reliable applications. Although their development is facilitated by toolkits (e.g. the ISIS toolkit [BiJ87]) which provide the basic building blocks of fault tolerance, the programmer still requires precise knowledge of the concepts implemented by these toolkits in order to use them successfully. Even with the help of toolkits, fault tolerance policies and mechanisms still have to be programmed explicitly as part of the application code. This close interrelation of the fault tolerance mechanisms and the application code has two severe drawbacks:

- Distributed fault tolerant applications become much more difficult to test because the implemented fault tolerance concepts have to be taken into account when selecting proper test scenarios. Often the failure handling parts of such an application are particularly difficult to test as subtle combinations of failure situations have to be simulated.
- Once the fault tolerance policy is chosen, the application is restricted only to this choice. Switching to a different policy (e.g. from passive replication by a primary-backup scheme to active replication) is usually infeasible without a complete application redesign.

The second argument is even more important if fault tolerance concepts are provided directly by the underlying operating system. In this case, all applications are restricted to only these mechanisms, regardless of their specific reliability requirements.

In this paper we present an approach to support programmers which are not familiar with fault tolerance concepts in building reliable applications. Our approach aims at making all reliability mechanisms application-independent and completely transparent for the application. With this approach the application programmer is relieved from implementing any fault tolerance related mechanisms by himself. Applications can thus be developed without having fault tolerance in mind, and all mechanisms needed to achieve and preserve the capability of tolerating system failures are added to a non-fault-tolerant application automatically without any changes of the program code. These reliability mechanisms are encapsulated within a separate layer which is provided to the application in form of a library. Different fault tolerance strategies can be implemented in different libraries, provided that the interface for the application does not change.

With this concept our approach offers a very flexible way of choosing various replication policies and mechanisms by simply selecting the corresponding library and linking it to the application code. In addition, testing of an application is substantially simplified. It is sufficient to test the non-fault tolerant application before linking it with one of the fault tolerance libraries, so that the code used to provide the reliability properties does not have to be considered in the test scenarios. Hence, testing can be restricted to the (non-fault-tolerant) functional part of the application. Failure cases may be disregarded, since these cases are handled within the ft-layer.

2. System model

For our approach we have chosen a very general computational model in order to make our concepts applicable for a wide range of systems. Our model is similar to the ones used e.g. in Eden, CONIC, Mach or Chorus. The computational entities are teams of processes which share a common address space. This common address space together with synchronization primitives based on semaphores can be used for efficient process communication within a single team.

Processes of different teams can only communicate with each other by message exchange. A message is sent to a single port (point-to-point message) or a port group (multicast message). Each port is owned by the team by which it was created, and only this team may receive messages from the port. A port group may comprise an arbitrary number of ports. The communication subsystem is assumed to provide an unreliable message transmission service, i.e. messages may be delayed or lost sporadically. However, if a message arrives at its recipient, its contents is uncorrupted. Based on this datagram service we assume the existence of a more reliable communication service according to a remote procedure call protocol which guarantees that a message is delivered at most once.

Teams are the unit of distribution and failure (i.e. all processes of a team execute on the same node, and a team failure causes all processes of the team to fail). Furthermore, it is assumed that teams fail silently, i.e. that they exhibit crash failure semantics [Cri91].

With our system proposed in the sequel we can tolerate up to *n* independent failures (both team failures and message losses) before the system has recovered from these failures and regains its capability to handle subsequent failures correctly. The value of *n* can be specified before the service is initially configured.

3. System architecture

We describe our system architecture in terms of a client-server model which is based on a modern micro-kernel. In this model, on each node of the network the operating system kernel provides a functional interface for the client and server applications. Interaction between these applications is only possible by calling the communication facilities offered by the kernel (Fig. 1).

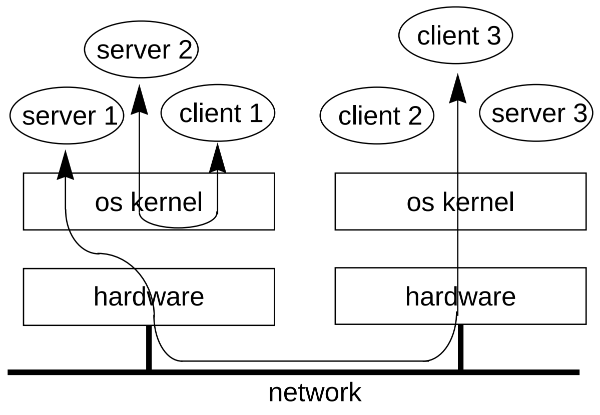

Fig. 1: Client-server system architecture

In order to make a service resilient to system component failures, several servers providing this service are executed on distinct nodes of the system. Such a set of servers cooperating in a single service is called a *server group*. In case that one or more of these servers fail, one of the remaining group members can still provide the service to the clients.

Our major goal was to make all fault tolerance mechanisms transparent to the application programmer. We therefore have developed a fault tolerance layer (ft-layer) which is inserted between each member of the server group and its underlying operating system kernel. This layer provides the same functional interface to the application as the kernel, but internally enhances the semantics of the kernel functions, depending on the fault tolerance properties implemented by the layer (Fig. 2). Since the functional interfaces of the ft-layer and the original operating system kernel are identical, the ft-layer is completely invisible for the application and hence can easily be substituted by an ft-layer implementing a different reliability concept.

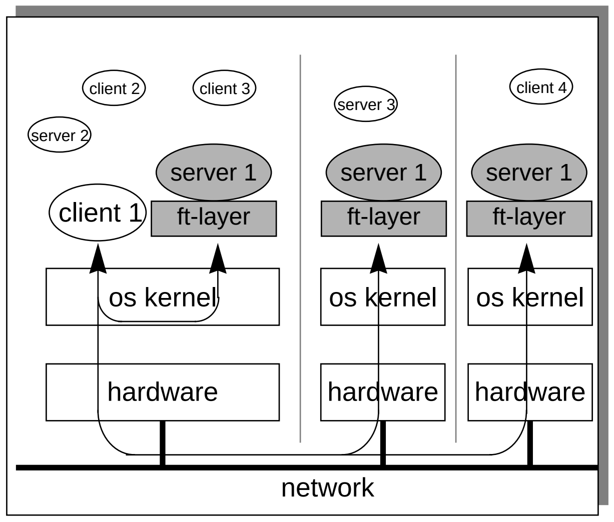

Fig. 2: Fault tolerant server group

The ft-layer encapsulates all basic building blocks which are necessary for a fault tolerant application. These building blocks are:

- **Service configuration.** The ft-layer is responsible for the initial configuration of the service according to configuration-specific parameters (e.g. the number of faults to tolerate during a specific time interval). This includes the automatic replication of a server and the mapping of the replicas on distinct nodes of the network.
- **Group member surveillance.** All server replicas form a group and are kept under surveillance to detect failures as soon as possible. If group membership changes, membership information is updated and distributed among the remaining group members consistently. This guarantees that each group member has the same membership view.
- **Service reconfiguration after failures.** In case that a failure has been detected, recovery actions are invoked in order to reestablish the desired degree of replication. The service will be reconfigured automatically by starting a new server replica for each failed group member on an operational node. The new replicas are initialized properly and included into the server group.
- **Replica management.** Coordination, synchronization, and management of the server replicas is handled within the ft-layer in order to guarantee service consistency conditions induced by the implemented replication strategy. Note that this is done without semantic knowledge of the application.
- **Input message distribution.** Input messages are distributed to all members of the server group. If necessary, an identical order of message delivery is constructed for each recipient using an atomic and order-preserving multicast protocol.
- **Output message selection.** If more than one output message is generated by the server group, the ft-layers of the server replicas have to agree upon one of these messages being passed on to its destination. All other messages are discarded in order to avoid message duplication.

Configuration, failure detection and failure recovery are activities of the ft-layer which must be available at any time. This requires the ft-layer to be an active component of the server team. For example, it is essential that the failure of a server group member is detected as soon as possible, and recovery from this failure is performed instantly in order to reestablish the required degree of replication (and hence the degree of fault tolerance), even if the application layers of all servers are currently inactive (e.g. waiting for a new client request). Therefore, the ft-layer does not only enhance the semantics of the functions provided by the underlying kernel, but also comprises additional processes which handle all activities concerned with the fault tolerance mechanisms. Internal state information of the ft-layer is kept within a memory region which is separated from the address space of the application layer. This separation is caused by technical reasons which will be given below.

4. Inside the fault tolerance layer

Although the tasks of the ft-layer have been clearly defined in the previous section, its internal structure strongly depends on the replication policy and the mechanisms chosen to provide the desired reliability. In the sequel we consider two example layers implementing different policies. In the first layer we are concerned with a passive replication scheme consisting of a primary server and an arbitrary number of backup servers. In the second example, we describe an ft-layer which provides mechanisms for the replication of actively executing servers.

4.1. Fault tolerance layer for passive replication

In the passive replication scheme a fault tolerant service is initialized by starting a single server which automatically becomes the primary server. Before processing the first client request the primary configures the service by starting a group of *n*-1 identical backup servers on distinct nodes. The replication degree *n* is a configuration parameter which can be specified at service invocation time. It implicitly defines the maximum number of failures which the system is capable of tolerating before it has completely recovered from these failures. In the case of the ft-layer for passive replication, the maximum number of tolerated failures is *n*-1.

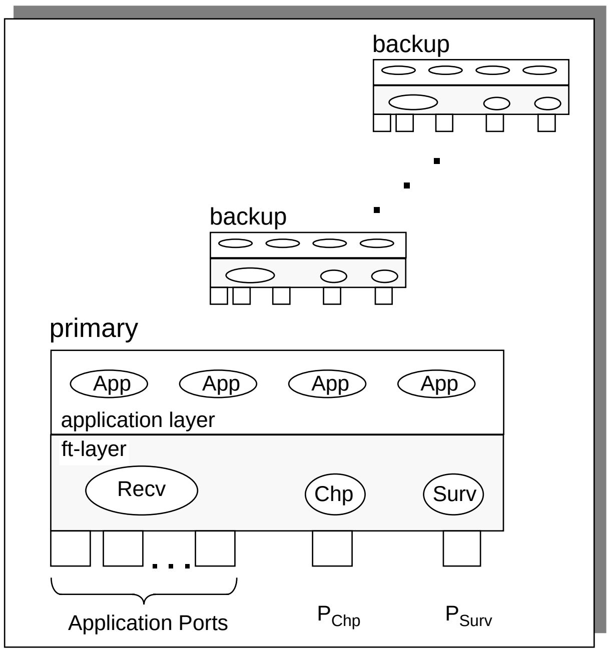

Fig. 3: Architecture of the Passive Replication Layer

The ft-layer of each server group member contains a surveillance process *Surv* (Fig. 3). The surveillance processes communicate with each other using the surveillance ports P_{Surv} to detect member failures. We use a protocol similar to the attendance list membership protocol proposed by Cristian [Cri91a]. The server group is arranged in a virtual ring on which messages containing the current membership list circulate from one group member to the other. Initially the execution of the protocol is controlled by the primary. A group member failure is suspected by all surviving members if the membership message fails to arrive after a watchdog timer has expired. In this case, an agreement protocol [Bec91] is started to select a new primary server (or to confirm the old primary if it is still operational). This server is responsible for collection and distribution of the new membership information and restart of the surveillance protocol.

After a group member failure a new backup replica is started and initialized with the state of its creator. If possible, this is done by one of the remaining backups to minimize the impact of recovery on the service response time.

In order that a backup server which takes over after a primary failure need not reexecute the whole computation of the primary, the states of the backups are updated by the primary at certain instances of time. For this purpose, the checkpoint process *Chp* of the primary's ft-layer sends a checkpoint to the P_{Chp} ports of all backup servers (all P_{Chp} ports are included in a port group to allow multicast communication).

In the primary, the *Chp* process first suspends all application processes and transfers those parts of the application state which have changed since the last checkpoint into a checkpoint message buffer. The decision which part of the state has changed is based on the value of a *dirty-bit* maintained by the memory management unit (mmu) for each memory page. All dirty-bits are cleared during a checkpoint. If a memory page has been written since the last checkpoint, its dirty-bit was set by the mmu, and the page will be transferred to the backups.

The ft-layer maintains its own state information which usually differs between primary and backups. Therefore, this part of the replica state must not be transferred by a checkpoint. If ft-layer state information were stored on a memory page of the application layer, checkpointing of this memory page would destroy ft-layer information in the backup servers. Therefore, the virtual team address space is separated into an application part and a part which is private to the ft-layer. A checkpoint only transfers memory pages associated with the application layer.

In addition to the address space contents, context information of the application processes (register contents, stack pointer, program counter, status information, etc.) is obtained and stored in the checkpoint message buffer. Finally, the states of the application ports (i.e. information about the messages pending at these ports) are copied into the buffer, and the application processes are resumed.

Subsequently the *Chp* process multicasts the stored checkpoint information to all backup replicas. Due to its size a checkpoint must be split over several messages. A message sequence number and a two-phase-commit protocol are used in order to guarantee that a checkpoint is either installed entirely at all destination sites so that the backup states are consistent, or it is not installed at any of them (in which case the consistent state of the previous checkpoint is preserved). Only if all backup servers confirm the correct reception of the checkpoint, a commit message is sent to the backups. Otherwise an abort message is sent.

In each backup replica the application processes are suspended as long as the backup does not take over as new primary. However, the processes of the ft-layer are active. The *Chp* process is waiting to receive a new checkpoint at the P_{Chp} port. Upon reception of a checkpoint message, the message contents is buffered. If a checkpoint message is missed (which is noticed by a gap in the sequence numbers) the message is requested from the checkpoint sender. As soon as all messages have been received, a confirmation message is returned, and the *Chp* process waits for a corresponding checkpoint commit message. Upon reception of this commit message the new server state is installed by extracting the state information from the checkpoint buffer and copying it into the application address space. If an abort message is sent, the checkpoint buffer is cleared.

In non-deterministic applications structured according to a general computational model such as ours it is an important problem to decide when a checkpoint must be issued. To avoid inconsistent service behavior, all effects caused by the repetition of statements after a primary failure (such as messages already sent by the primary) must remain invisible for other teams. In addition to that, the backup which takes over as new primary must perform the same state transitions as the failed primary since its last checkpoint to reach the same *externally visible* state, i.e. the state which is seen by clients communicating with the service and by other teams in the service environment. In particular, state transitions which have been performed as a result of a non-deterministic decision must be repeated in the same way by the backup in case of a recovery.

In our ft-layer for passive replication inconsistent service behavior is avoided by a method called *systematic checkpointing*. In this method, a checkpoint will be issued whenever the externally visible state changes [Bec92]. In our model this is only the case if

- a new port is created or an existing port is deleted, or
- a message is sent by the primary.

The corresponding kernel functions of the ft-layer which overload the functions of the original kernel invoke the checkpointing mechanism automatically when called by the application.

Clients issue service requests by sending request messages to the application ports of the server group. In order that all server replicas receive the request messages, the corresponding user ports of each replica form a port group, and request messages are sent using a reliable multicast protocol. We do not require preservation of message order, so the multicast protocol used here can be very efficient. The receiver process *Recv* of the primary ft-layer passes the request messages on to the application processes, while in the backup servers the messages are buffered in the ft-layers until either they can be discarded after a subsequent checkpoint, or one of the backups takes over.

Note that the order of messages need not be identical in the primary and the backups, since each non-deterministic decision caused by the message order will be checkpointed as soon as it becomes visible outside the primary server. Therefore, the checkpointing mechanism enforces consistent message order implicitly by selecting which messages may be discarded at the backup sites, and which may not.

4.2. Fault tolerance layer for active replication

Determinism considerations

Applications which are replicated and executed in parallel usually are required to be deterministic. However, guaranteeing determinism even for applications based on a single-process model is very difficult and expensive in practice. In particular, the operating system kernel on which an application is based produces some application input (e.g. in form of return values of kernel functions). This information transfer from the kernel to the application is independent of the system model, and therefore has to be considered as a source of non-determinism for all applications. For example, consider a kernel call which is used to allocate a resource. While some of the application replicas allocate the resource successfully, other replicas may not be successful due to a temporary shortage of the resource on their local nodes. In each replica the information about the success of the kernel call is transferred to the application layer which may use this information to determine the next state transition. The applications obviously are non-deterministic, making different state transitions within the group possible.

It is therefore necessary to either consider non-deterministic applications in general for active replication, or enforce determinism by a very restrictive model and handle each operating system kernel output as application input. The latter requires extremely close synchronization of the replicas (at the base of each operating system kernel call returning a result), thus causing substantial overhead and reducing service throughput.

Our approach can handle non-deterministic applications without requiring such a close synchronization of server replicas. We make the assumption that a service is *intended* to produce deterministic output (which is much weaker than the assumption that a service exhibits deterministic *behavior!*) and take measures to detect and mask state divergences of group members if output messages diverge. A good example application is a name service which maps logical service names to physical addresses at which the services are available. Since the mapping of names to addresses is unique, service requests for information retrieval are deterministic. A client registers a new service by specifying a service name and the associated service address, so service registration per se is also a deterministic operation. Only if two different clients try to register a new service at the same time, and both service requests are processed in parallel by different processes in each replica, a conflicting decision can be made by the group members if the same service name is to be used for two different service addresses. Although this case is rare enough so that a state divergence based on this conflict is very unlikely, care must be taken to resolve this conflict and to hide the effects of this potential non-determinism from the service environment.

In our approach group member synchronization is required only if the externally visible state of a service changes, i.e. if a message is sent by the service or if the service port interface changes. In this case the ft-layers interact to agree on the same output of all group members. Members which disagree will be enforced to adopt the state from one of the agreeing members. To minimize the impact of synchronization and masking of non-deterministic decisions on the service response time, we eliminate sources of non-determinism where possible, thus reducing the probability of their occurrence. As a simple example, we take care that process identifiers are identical for corresponding processes in all replicas by using virtual identifiers reflecting the dynamic creation history of the process.

Layer architecture

In the active replication scheme service configuration is done by the server which is started first. It creates n-1 identical server replicas on different nodes of the network to establish the server group. As in the case of passive replication, a surveillance mechanism is initiated to discover group member failures. This surveillance mechanism is incorporated into an atomic multicast layer which is used by clients for communication with the service and for coordination purposes within the server group. The atomic multicast layer has been developed especially for the use within a group of replicated servers and perfectly meets the requirements of our ft-layer [BeG93]. The communication protocol executed in this layer constructs a consistent order of messages at all ports of each port group by a virtual token ring connecting all group members. The token is also used to detect member failures and to distribute membership information to all group members consistently. Therefore, an additional surveillance protocol is superfluous. The *RecvSurv* process of the ft-layer is responsible for the protocol execution and group member surveillance (Fig. 4).

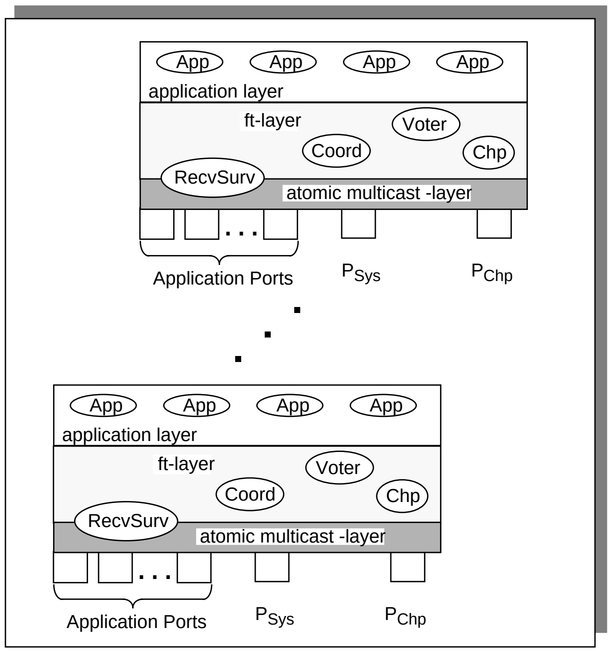

Fig. 4: Architecture of the active replication layer

If a server failure is detected, the atomic multicast layer generates a crash event. Crash event messages are disseminated among the server group using the atomic multicast protocol and therefore arrive at each group member in a consistent order. These messages are delivered at the service system port P_{Sys} of each group member. Upon reception of a crash event message the *Coord* processes of all ft-layers agree on one server which is responsible for creating a new replica on an operational node. Surveillance of the new replica is done by its creator until it joins the virtual ring, causing the atomic multicast layer to generate a join event message. Like crash events, the join messages are ordered consistently so that each group member receives the same sequence of changes in the group membership.

The selection of output messages is done by the *Voter* process in the ft-layer. Since in our system model each process of a team creates an independent stream of output messages, the *Voter* processes consider output messages for each process separately. Our voting protocol is reminiscent of the 'Competitive Validate-before-Propagate Mode' protocol proposed in [Pow91]. Instead of being sent directly by an application process, an output message is passed on to the local *Voter* process. Before the message is actually sent to its destination, the *Voter* processes send a claim message to the P_{Sys} port group using the atomic multicast protocol. This claim message contains a signature of the user message. If a majority of identical claim messages has been found, the ft-layers agree on one server replica which actually sends the message, while the other replicas discard it. Agreement on who is responsible to send the messages can be easily established based on the sequence of claim messages arriving at each group member in an identical order. After the message was sent, the success of this operation is disseminated among the group members by another atomic multicast. This confirmation is necessary in order to avoid messages to be sent twice after a failure.

When selecting an output message from the messages produced by all server replicas in response to a client request, it must be guaranteed that the sequence of output messages is consistent. This means that the same sequence of output messages can be generated by a single server provided with the same stream of input messages. If all server replicas have the same internal state before processing a client request, then the result message produced by any replica may be chosen for being returned to the client. However, if the server replica states have diverged, then the choice of any subsequent output message must be restricted to those servers which have agreed with the former output message. The choice of a different message might result in a message sequence which cannot be produced by a single server, thus contradicting the consistency requirements of the service output.

Once a server replica has produced a message which is excluded from this choice, the state of this server may be inconsistent to the sequence of state transitions performed up to now, and hence must not be considered any more for further processing of client requests. Therefore, the degree of replication and hence the fault tolerance degree are reduced.

To compensate this reduction of redundancy we enforce each group member which does not agree with a selected output message to adopt the internal state of an agreeing server replica. In such a case the *Voter* process of one of the agreeing replicas issues a checkpointing request to the *Chp* process, specifying which group member states have to be adjusted. The *Chp* process then suspends the application processes, copies the application layer state into a checkpoint message buffer, and resumes the application processes. Then a checkpoint request message is sent to all disagreeing group members. Their *Chp* processes suspend the application processes and wait for the checkpoint messages to be sent. After reception of all messages and successful installation of the new replica state, the application processes are resumed.

This adjustment of server replicas to a consistent service state also works if the *Voter* processes cannot find a majority of output messages. In principle, any output message can be selected since our checkpointing mechanism guarantees that after each output the service is in a consistent state. However, if a majority of server replicas already agree on a message the overhead of checkpointing is minimized.

Since checkpointing a server state reduces the performance benefits of active replication, the ft-layer for active replication is suitable for server applications which exhibit (externally visible) non-deterministic behavior only in rare cases. If such a behavior occurs more often, the passive replication scheme should be used instead, since non-deterministic decisions are handled implicitly there by the checkpoints.

We have chosen the potentially non-deterministic team model to demonstrate the general applicability of our approach even for non-deterministic applications. However, the overhead caused by the ft-layer can be substantially reduced if determinism of applications is guaranteed (e.g. by a restricted system model, deterministic kernel output handling, etc.). If replica states cannot diverge and all group members are provided with the same sequence of input messages, a simplified voting strategy can be applied, and the voter processes may select an arbitrary replica which actually sends the message (e.g the fastest one producing the output). There is no need to collect a majority of identical messages, and the checkpointing mechanism becomes completely obsolete.

5. Performance considerations

Our performance considerations are based on the assumption that a service is accessed by a client using a remote procedure call (RPC) like communication pattern. Therefore, the delay of the service response message imposed by the ft-layer should be minimized.

5.1. Passive replication layer

When using a passive replication scheme, in normal operation mode (i.e. the failure-free cases) a checkpoint is required whenever the port interface of the service changes or an output message is sent by the primary server replica. To hide the impact of the checkpoint caused by the reply message of the RPC, it would be desirable to send this checkpoint after the reply. However, if the primary fails between sending the reply message and the checkpoint transfer, the backup which takes over may produce and send a different reply message due to non-deterministic decisions, which contradicts the demand for service consistency. This can only be avoided if the checkpoint is sent immediately *before* the reply message. Then the checkpoint guarantees that the same reply message will be repeated, should the primary fail. To minimize the probability of generating such a message duplication, a confirm message is sent to all backups to notify them about the success of the send operation.

As a consequence, in a passive replication scheme the client application cannot be prevented from noticing the effects of checkpointing if the application may generate non-deterministic output, since a checkpoint must always be sent between the reception of the service request and sending the service reply message.

The cost of a checkpoint itself strongly depends on the service application structure. For instance, each process of the primary server team has its own stack, which must be included in the checkpoint if the process has been active and changed the stack. Therefore, the size of a checkpoint depends on the number of processes in the primary server. Similar considerations apply to the number of application ports. For each port, state information has to be collected and sent along with the checkpoint. So the checkpoint size also grows with the number of ports. Finally, another important factor for the amount of checkpoint data is the size of a single memory page which determines the granularity at which changes in the address space must be transferred to the backups.

Even if the checkpoint data are limited to those pages which have been modified since the last checkpoint, the total amount of a checkpoint is significant. It is therefore essential to apply compression methods (like run length encoding or methods using shadow pages) before a checkpoint is split into messages and sent over the network. The quality of the applied method is a dominant factor for the efficiency of a passive replication scheme.

In case of a group member failure, the impact of recovery on the service response time depends on whether the primary or a backup has failed. If a backup has failed, one of the remaining backups is responsible for creating a new replica and initializing it with the last checkpoint, so that a client will not notice any delay caused by this service reconfiguration.

If, however, the primary server fails, the client will not get any service response before the failure has been detected by the backups and a new primary has been selected. This time is largely determined by the setting of the watchdog timer used to detect group member failures. After the failure has been detected, the election of a new primary can be performed by n+1 multicasts [Bec91], where n is the number of the surviving group members. The winner of the election immediately continues processing the pending service requests.

5.2. Active replication layer

The advantage of active server replication lies in the fact that in normal operation mode expensive checkpointing is not required. Therefore, the delay in service response time as seen by the client is much smaller than in passive replication schemes. When using active replication, the performance overhead for a typical RPC-like client-server communication pattern is mostly influenced by three causes. First, the service request message must be disseminated among all server replicas by an atomic and order-preserving multicast protocol to guarantee service consistency. Second, after the service request has been processed by the replicas, they have to agree on one reply message to be sent back to the client, and on a server which actually sends the message. This is usually done by a voting algorithm which handles different messages produced by the server replicas. Finally, the server which has sent the message must report the success of this operation to all other group members to avoid message duplication in case of a failure.

If servers have produced different output messages, those servers which do not agree with the selected message must adopt a new state from one of the other server replicas. As in the case of passive replication, this is done by a checkpoint which is sent from one of the agreeing replicas to those who disagree. The size of such an adoption checkpoint depends on the application structure. However, although these checkpoints are expected to occur very infrequently, they have to transfer the complete team address space.

In our prototype implementation of the ft-layer for active replication, we have measured the average service response time for a null RPC, using our atomic multicast protocol described in [BeG93]. The implementation was done on top of the experimental MOSKITO operating system kernel [NeG90]. The experiments have been performed on SUN 4/25 workstations interconnected by a 10 MBit/s Ethernet. Sender and receiver teams were all running on distinct nodes, so no local communication was involved.

In these experiments our ft-layer has shown a total service delay time of less than 36 ms for a server group of two replicas, and less than 50 ms for a group consisting of three servers. Our atomic multicast protocol delivers a broadcast message after about 10 ms in a ring with two receivers, and after about 13 ms in a ring with three receivers. Voting as well as the distribution of the acknowledgment after successfully sending the reply message are done using the atomic multicast protocol, so that three successive atomic multicast operations influence the service response time. The numbers given above show that the additional overhead of the ft-layer without the time used by the atomic multicast protocol is in the range of 6 to 11 ms for a group of two or three receivers, respectively.

6. Related Work

Several techniques for achieving fault tolerance have been described in literature which are similar to ours. In the ISIS project [BiJ87] a toolkit for the construction of fault tolerant applications has been developed. This toolkit comprises a set of atomic broadcast protocols which differ in their degree of synchronization of the receivers and ordering semantics of the messages. A coordinator/cohort scheme [BiJ87, BCJ90] has been proposed which is based on these atomic broadcast protocols. In this scheme one member of a group of communicating processes is selected to be the coordinator of the group which is responsible to actually perform a computation on client request. The other processes serve as passive cohorts which are ready to replace the coordinator in case of a failure. Once a coordinator has been selected for a computation, it is impossible for a new member to join the cohorts. Therefore, the degree of fault tolerance decreases with each failure during a computation. In our system, the ft-layer also is responsible to reestablish the degree of fault tolerance specified at configuration time by creating a new server replica for each failed one. The aim of the ISIS toolkit was to encapsulate all fault tolerance *mechanisms* and make them available to the application programmer who still is in charge to implement the fault tolerance *policies*. Our ft-layer includes both mechanisms and policies and thus achieves complete transparence of the fault tolerance property of a service.

In the Delta-4 project [Pow91] several different fault tolerance techniques have been investigated. The architecture of fault tolerant applications is similar to the one proposed in this paper. Each server replica is associated with a “replica coordination entity” which is responsible for the coordination and management of the replicas. In the passive replication technique described in [SpB89] the application programmer has to specify recovery actions which are to be taken if the primary has failed while interacting with its environment other than by message exchange (e.g. by input / output). Our method of implicit checkpoints taken after each output renders additional recovery actions unnecessary. The second technique used in Delta-4 is called the leader-follower model [BHV90]. A software component is replicated in *n* identical copies which are all active. One of the replicas, the leader, takes all (potentially non-deterministic) decisions before propagating them to the other replicas, the followers. By exchanging synchronization messages it is guaranteed that the followers always execute at least one step behind the leader. Finally, the Delta-4 project also was exploring an active replication scheme [CPR92]. However, in their scheme, the replicated software components are required to be deterministic.

For the Conic system [LKE86] a hot standby scheme is proposed which is similar to our passive replication technique. In this approach, several copies of an application program are distributed on distinct processors. One copy is active, processing service requests, while the other copies are waiting to receive checkpoints from the active one. The detection of failures and the transfer of checkpoints is done by hot standby managers which are associated with each program copy. Service reconfiguration after a failure is performed by a configuration management service consisting of a configuration manager and a status collector.

7. Conclusions

With our concepts we have shown that the encapsulation of fault tolerance mechanisms in a layer between the operating system kernel and the application has two main advantages: First, the reliability mechanisms are provided in a completely transparent and application-independent way. Therefore, even complex applications can be developed regardless of any aspects of fault tolerance, and testing of these applications is simplified substantially. This minimizes the development time for fault tolerant applications drastically. Second, replication policies and reliability mechanisms can be easily changed without application redesign by simply choosing a different ft-layer.

In case of active replication, it is necessary to consider non-deterministic applications, since even in very restrictive system models operating system kernel output may lead to non-deterministic decisions, causing the states of server replicas to diverge. We have shown that it is sufficient to detect such state divergencies only if they are visible for the service environment. In this case, disagreeing server replicas are forced to adopt a new state from the correct ones, so that the total degree of fault tolerance is not reduced.

Acknowledgments

We would like to thank A. Beranek and the unknown referees for their valuable comments on the manuscript of this paper. We are also grateful to R. Neumann and M. Stopp for doing the prototype implementation of the ft-layer examples.

References

[BCJ90] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, M. Wood: The ISIS System Manual, Version 2.1, *Cornell University, Ithaca, 1990.*

[BHV90] P. A. Barret, A. M. Hilborne, P. Veríssimo, L. Rodrigues, P.G. Bond, D.T. Seaton, N.A. Speirs: The Delta-4 Extra Performance Architecture (XPA), *Proc. 20th Symposium on Fault Tolerant Computing Systems, pp. 481—488, 1990.*

[Bec91] T. Becker: Keeping Processes Under Surveillance. *Proc. 10th Symposium on Reliable Distributed Systems, pp.198-205, 1991.*

[Bec92] T. Becker: Transparent Service Reconfiguration After Node Failures. *Proc. 1st Intl. Workshop on Configurable Distributed Systems, pp. 212-223, 1992.*

[BeG93] T. Becker, K. Grieger: An Efficient Atomic Multicast Protocol for Client-Server Models. *Proc. 7th Intl. Parallel Processing Symposium, pp. 816-823, 1993.*

[BiJ87] K. Birman, T. Joseph: Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems, Vol. 5, No. 1, pp. 47-76, 1987.*

[CPR92] M. Chérèque, D. Powell, P. Reynier, J. Richier, J. Voiron: Active Replication in Delta-4, *Proc. 22nd Symp. on Fault-Tolerant Computing Systems, 1992.*

[Cri91] F. Cristian: Understanding Fault-Tolerant Distributed Systems. *Communications of the ACM Vol. 34, No. 2, pp. 56-78, 1991.*

[Cri91a] F. Cristian: Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems. *Distributed Computing, Vol. 4,* pp. 175-187, 1991.

[KoT87] R. Koo, S. Toueg: Checkpointing and Recovery-Rollback for Distributed Systems. *IEEE Transactions on Software Engineering, Vol. SE-13, No. 1,* pp. 23-31, 1987.

[LKE86] O. G. Loques, J. Kramer, C. Eng: Flexible Fault-Tolerance for Distributed Computer Systems, *IEE Proceedings, Vol. 133, Pt. E, No. 6,* pp. 319—332, 1986.

[NeG90] J. Nehmer, T. Gauweiler: Design Rationale for the MOSKITO Kernel. *in: T. Härder, H. Wedekind, G. Zimmermann: Entwurf und Betrieb verteilter Systeme, Informatik Fachberichte Vol. 264, Springer Verlag,* pp. 91-109, 1990.

[Pow91] D. Powell (ed.): Delta-4: A Generic Architecture for Dependable Distributed Computing. *ESPRIT project 2252 Research Report, Springer Verlag,* 1991.

[SpB89] N. A. Speirs, P. A. Barret: Using Passive Replicates in Delta-4 to Provide Dependable Distributed Computing, *Proc. 19th International. Symposium on Fault Tolerant Computing Systems,* pp. 184-190, 1989.