An Approach to Transparent Fault Tolerance for Client-Server Models

Thomas Becker

October 1992

Technical Report
University of Kaiserslautern,
P.O. Box 3049, D-6750 Kaiserslautern, FRG
email*: tbecker@informatik.uni-kl.de*

Abstract

We describe a technique to make application programs fault tolerant. This technique is based on the concept of *checkpointing* from an active program to one or more passive backup copies which serve as an abstraction of stable memory. If the primary copy fails, one of the backup copies takes over and resumes processing service requests. After each failure a new backup copy is created in order to restore the replication degree of the service. All mechanisms necessary to achieve and maintain fault tolerance can be added automatically to the code of a non-fault tolerant server, thus making fault tolerance completely transparent for the application programmer.

1. Introduction

One of the most desirable properties of a distributed computing system is its potential capability of tolerating system component failures. In an idealistic view, if a component of a fault-tolerant system fails, another one takes over and resumes processing. This system reconfiguration would ideally be completely invisible from outside the system, and hence possible without additional interaction of a human operator.

A distributed, client-server structured system commonly comprises services which must be highly available. A complete breakdown of such a service causes other services to fail in consequence (or at least substantially reduces the quality of the provided service). A good example for a service with such availability requirements is the *name service* which maps logical service names to physical addresses to which service requests can be sent. If the name service fails, other services whose logical names are known become inaccessible because their physical addresses cannot be obtained.

Several basic techniques to achieve a fault-tolerant system behaviour are well known. In one of these techniques, the state of a program is transferred to non-volatile storage periodically. If the program (or the computer node on which it executes) fails, a new copy of the program is started and initialized with the last recorded state of the failed program. The new copy can then continue processing with a valid intermediate state without having to re-execute the complete statement sequence from the very beginning of the program. This method is commonly known as *checkpointing* or *cold stand-by technique;* the transfer of a program state is called a *checkpoint*. Depending on the granularity of the checkpoints (i.e. the time between two successive checkpoints) and the cost of creating and initializing a new program copy, the time which is necessary to recover from a failure and continue processing may be considerable. The service provided by this program appears to be unavailable during this recovery phase.

If unavailability of a service during recovery and reconfiguration of the system cannot be accepted, a different technique for tolerating failures must be used which is often termed *active replication* or *replicated execution* of a program [1]. In this approach, several identical replicas of a program execute in parallel on different computer nodes. If one of the replicas fails, the service will still be provided by the remaining replicas without notable recovery delay. However, to ensure that all replicas behave in a consistent way (i.e. produce the same results), specific synchronization measures have to be taken. In addition, while a program using a checkpointing strategy only requires a single processor, replication of a program is only reasonable if all replicas are located on distinct processors to prevent several of them from being lost after a single computer node failure.

In our *warm stand-by approach* we adopt advantages from both basic techniques. Our technique is based on the idea that a service is provided by several identical servers (i.e. replicas of an application program) from which only one (the *primary*) actually processes service requests from clients while the others (the *backups*) serve as an abstraction of stable memory. According to the cold stand-by approach, the primary periodically sends checkpoints to the backup servers. In case the primary fails, one of the backups takes over as the new primary using the state represented by the last successful checkpoint. Since there is no need to create a new server copy and initialize it with a state to be read from the disk before the service can be continued, the time during which the service appears unavailable for a client is reduced substantially.

An important concern of our approach is to make the fault tolerance concepts *transparent*, both for the designer of a service and for clients using the service. This allows an application program to be developed without having fault tolerance in mind. The mechanisms needed to achieve and maintain fault tolerance will be added automatically to the non-fault tolerant program code.

In Section 2 of this paper we describe the computational model for which our method has been developed. Section 3 gives an overview of a fault tolerant service architecture. In Section 4 we show how existing server code can be augmented with fault tolerance mechanisms automatically. Section 5 deals with implicit checkpoints. Failure handling and recovery actions are discussed in Section 6. Finally, we compare our technique with other systems in Section 7 and subsequently conclude the paper.

2. Computational Model and Fault Hypothesis

2.1. The MOSKITO kernel as a construction base

Our approach s described in terms of a client-server model [2] in which computational entities exchange messages to invoke the processing of tasks by other entities and to return results. The experimental operating system kernel MOSKITO [3] implements this model and has served as a construction base for a prototype of our method.

The computational entities managed by the MOSKITO kernel are *teams*. A team is a collection of processes which share a common address space. Each process can be viewed as an autonomous thread of control. Teams and processes can be created and destroyed dynamically.

Due to the common address space processes within a team are able to communicate with each other using shared variables. The communication between processes of different teams is solely done by message passing. A process can send a message to a *port* which, like teams and processes, may be created and destroyed during run time. Each port has a system-wide unique identifier. A port is associated with the team from which it was created. Only processes of this team may receive messages from the port.

The MOSKITO kernel provides a multicast communication facility by the concept of *port groups*. A port group comprises an arbitrary number of ports which may join and leave the port group at system run time. A port can only be member in one port group. Messages with a port group as destination address will be sent to all ports which currently are member in the port group.

The send primitive provided by MOSKITO offers two different communication semantics. A *datagram* message is an asynchronous, notification-oriented message. The sender of a datagram message resumes processing immediately and is not informed about the success of the send operation (i.e. a datagram message is unreliable). When using the synchronous order-oriented message transmission scheme, the sender is blocked until the message was received at its destination and a process of the receiver team has sent a reply message. This communication scheme implements a *remote procedure call* (rpc). In case the message is multicast, the first reply message will deblock the sender process.

Our method requires a homogenous distributed system in which the nodes are interconnected by a local area network. It is assumed that MOSKITO runs on each of these nodes.

2.2. Fault Hypothesis

When implementing a fault tolerance mechanism, a careful consideration of the failure classes which are supposed to be handled by the mechanism is necessary. Cristian [4] has proposed a failure classification scheme which reflects the effects of failures as seen by the application software (which is typically structured according to the client-server paradigm). In this classification, a server is said to be correct if it behaves according to all its specifications.

A server which omits to respond to an input suffers from an *omission failure.* A *crash failure* has occurred if a server does not process subsequent requests any more. If the processing of service requests is not performed in a timely manner (i.e. the server response arrives at the client either too late or too early), this is called a *timing failure*. Finally, an *arbitrary failure (response failure)* is a failure which is not included in one of the failure classes described above. Examples for arbitrary failures are server responses containing wrong information, or incorrect server state transitions. These failures are commonly known as “Byzantine failures” [5].

In general, failures of type *arbitrary* are very difficult to handle [6]. Since in our communication subsystem message transmission time is unpredictable, we restrict our failure model on the class of timing failures (which includes crash and omission failures). Messages may be lost, but no message delivered at its destination will be corrupted. FIFO ordering of messages is guaranteed between two communicating processes.

Teams may fail at any instance of time (e.g. due to a crash of the node on which they have been executing). A fail-silent behaviour is assumed, which means that every process in the team stops processing immediately after a failure, and no more messages will be sent.

3. Service Architecture

In our approach, a distributed fault-tolerant service consists of *k* identical servers running on distinct nodes of the computing system. The number *k* of replicas is a configuration parameter of the service which allows to balance the degree of fault tolerance with the overhead induced by the replication according to service specific needs.

One of these *k* server replicas is responsible for processing service requests and returning results to the calling clients. This server is called the *primary server*. The other *k*-1 backup replicas are passive in the sense that they do not process service requests.

As long as the primary server is alive, it sends checkpoints to all backups. A checkpoint contains only those parts of the primary state which have changed since the last checkpoint. Upon reception of a checkpoint, each backup server updates its local state, thus adopting the state of the primary when the checkpoint was issued.

An important concern of our method is that all aspects of the service architecture are hidden from the clients. Clients need not know about the replication degree of a service, or which of the servers currently is the active one. In particular, a client is not even aware of the fact that it communicates with a fault-tolerant service if the interface structure (i.e. the number of ports and the message protocols defined for these ports) of the fault-tolerant service is the same as the one of a single server.

In case of a server failure the service performs recovery actions and reconfigures itself. It is essential that this reconfiguration is completely invisible for clients. Therefore, the service interface and the addresses to which service request messages are to be sent may not change. However, if a backup server takes over as the new primary, its port identifiers will be different from the ones of the failed server. To avoid this change in the service interface we use the port group concept of MOSKTIO (Figure 1). For each service port created by the primary server a new port group is created with the service port as the first member. All backups create a new service port, too, and add their new port to the port group.

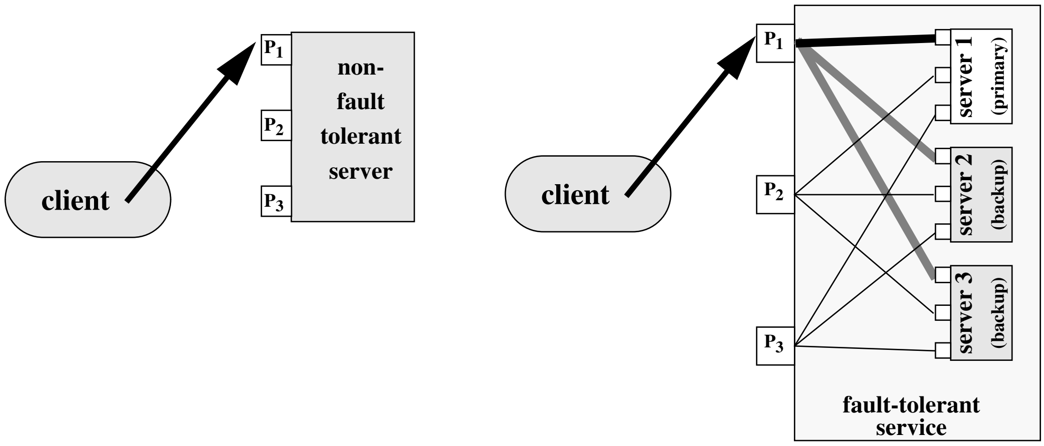

Figure 1: Interface structure of single and replicated servers

Client request messages are sent to the service port groups instead of ports. In case of a service reconfiguration, the identifiers of the port groups do not change, and the service reconfiguration is completely hidden from the client. Since the identifiers of MOSKITO ports and port groups have the same structure, clients need not distinguish between single and replicated servers.

4. Server Code Augmentation

With our method the development of fault tolerant service applications is substantially simplified because all fault tolerance components are encapsulated in a library package which is independent of the application code. Therefore, a service can be made fault tolerant by automatically augmenting the non-fault tolerant server code with functions from the library package. As illustrated in Figure 2, a Code Augmentation Tool performs the necessary changes of the server code to produce code for a distributed, fault tolerant service with equivalent functional behaviour, comprising k identical server replicas.

Each server replica is provided with a set of library functions which are necessary to achieve and maintain the fault tolerance properties of the service. One of these functions configures the service during its initialization phase. The primary creates k-1 backup servers on distinct computer nodes and assigns each of them a unique ranking number according to their creation order (i.e. the first backup server is assigned ranking number 1, the second backup gets ranking number 2, etc.). These ranking numbers define an order on the k server replicas which is used to

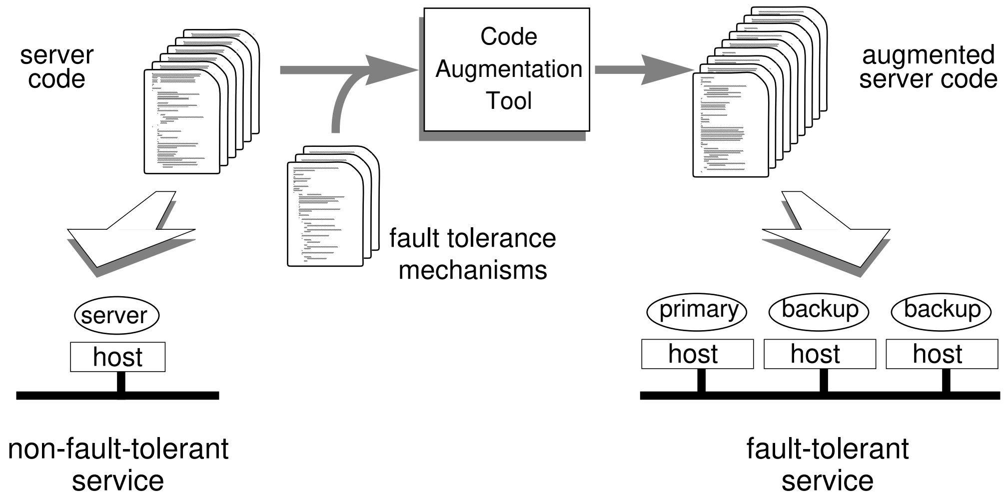

Figure 2: Automatic server code augmentation

arrange them to a logical ring. In addition, they will be used to determine the backup server which takes over if the primary fails, so that dynamic election of the new primary [7] is avoided.

Once the service is configured and initialized, client requests are processed by the primary. In order to preserve the availability of the service during its life time, server failures must be detected and the service must be reconfigured to recover from the failure. Server failure detection is done by a *neighbourhood surveillance protocol* proposed by Cristian [4]. In this protocol, *alive* messages are sent along the logical ring periodically. If these messages cease to arrive at a server a failure of its predecessor in the ring is concluded.

In Figure 3, the implementation of these concepts is sketched. For ease of presentation it is assumed that all service functions are available at a single service port P_s. The fault tolerance concepts within each server are implemented by three additional processes and two ports. The *trigger* and *watcher* processes execute the neighbourhood surveillance protocol. Each trigger process periodically sends an *alive* message to the surveillance port P_a of its logical successor. The watcher process controls the arrival of these messages and initiates recovery actions if the messages cease to arrive.

The *update* process and the port P_c are used for checkpointing. The primary issues a checkpoint by multicasting a message to the checkpoint ports P_c of all backup servers (a port group comprising all checkpoint ports is created during the initialization phase of the service). The message contains those parts of the primary

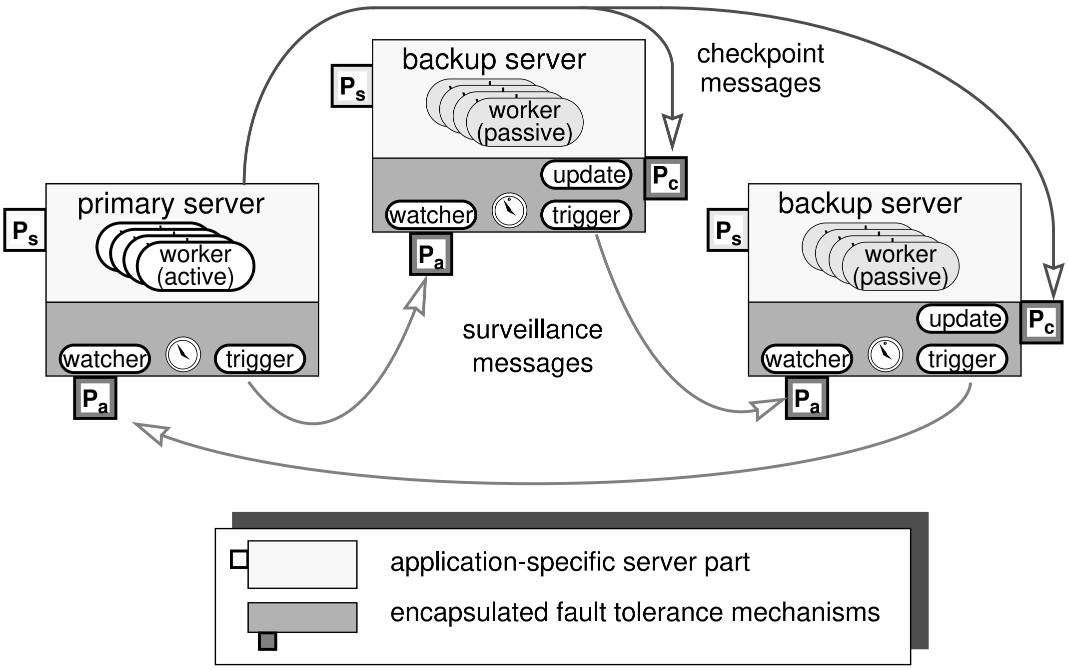

Figure 3: Architecture of a fault-tolerant service

server state which have changed since the last checkpoint. Upon reception of a checkpoint message the update processes change the local states of the backup servers according to the message contents. Note that in the primary server the update process and the checkpoint port are obsolete since the primary will never receive any checkpoints.

5. Implicit Checkpoints

Above we have described how checkpoints are received and processed in the backup servers. We now focus on the question when and how checkpoints are issued. Since with our method all fault tolerance mechanisms are transparent for the application program, the decision when to issue a checkpoint must be taken within the fault tolerance package, independent of the current server state.

The most obvious approach to issue checkpoints periodically is not feasible in our model because MOSKITO teams are potentially nondeterministic. This may lead to the following situation.

Assume that after a checkpoint was issued, the primary server executes a nondeterministic function and sends a message containing the result to its environment (e.g. to a client team). Before the next checkpoint is taken, the primary fails. After a recovery phase, the new primary takes over from the last checkpoint. The nondeterministic function is executed again, but now produces a different result.

The key problem in this scenario is that the client team knows about the result of the nondeterministic function executed by the failed primary, and hence about its state. However, the state of the new primary actually differs from the state of the failed one.

To avoid such an erroneous behaviour of a service, we introduce the notion of the *externally visible service state*, which is reminiscent of Schneider's *state machine approach* [8]. From a client's point of view, a service has an observable state. This state changes if

- the service communicates with its environment. Each message sent by the server contains part of the server state. Therefore, state changes become visible to the receiver(s) of the message.
- the interface structure of the service changes, e.g. if a new service port is created.

Whenever the externally visible service state changes, a checkpoint is issued to guarantee that all nondeterministic decisions taken by the primary are transferred to the backup servers as soon as they can be seen by other teams. The occasions at which all these state changes take place can be easily identified since they are results of operating system function calls. We therefore provide the servers with stub procedures which automatically issue checkpoints if necessary.

Since a checkpoint is invoked after each message sent from the primary server, re-sending of messages is avoided after the backup has taken over, and the backup server resumes processing from a state which reflects all the visible nondeterministic decisions taken by the failed primary. Checkpointing after service interface changes causes the backup servers to adopt the primary's new interface structure immediately. If, for example, the primary has created a new service port, an implicit checkpoint causes all backup servers to create a new port, too, and join all ports in a new port group. Thus, messages sent to the new port group will also arrive at the backup servers.

Message logging at the backup servers is implicitly done by message queues associated with the ports. Messages which have been received by the primary will be discarded by the backups. For this purpose, a message identifier list of those messages which have been received is sent to the backups along with each checkpoint.

6. Server Failure and Recovery

A server failure is detected by the watcher process of its ring neighbour. Since each server exhibits a fail-stop behaviour, no more “alive” messages will be sent to the neighbour surveillance port P_a after a server has failed. The watcher processes use a watchdog timer to control the arrival of the “alive” messages. If unreliable communication primitives are used, a loss of a message may cause a reconfiguration to be initiated which in fact is unnecessary. To reduce the probability of such a false alarm, a fixed number of successive message losses is allowed before a watcher process concludes that the predecessor server has failed. However, this comes with the price of an increased minimum time in which a failure can be detected.

Once a server has concluded that its predecessor has failed, recovery actions must be performed to reconfigure the service and to re-establish the logical ring which is broken due to the failure.

Depending on which of the servers has failed, different measures for service reconfiguration have to be taken. Let us first assume that only the primary has failed. In this case, the backup server with the lowest ranking number (i.e. the successor of the old primary in the ring) detects the failure and takes over immediately from the last checkpoint. To preserve the replication degree *k* specified in the configuration phase, a new server replica must be started on a computer node which does not yet host a replica and initialized with the last checkpoint. This is done by one of the backup servers so that the new primary is discharged from reconfiguration management and can resume processing service requests.

If one of the backup servers fails, reconfiguration of the service is limited to starting and initializing another server replica. In case the primary has detected the failure of its predecessor, one of the backups is ordered to manage the reconfiguration. Otherwise, the backup server which has detected the failure is responsible for creating the new replica.

Recovery from a single server failure does not cause a new primary to be elected dynamically because the order imposed by the servers’ ranking numbers can be used. However, if several servers fail at a time, service reconfiguration becomes a bit more complex.

Consider the case where the primary and its successor backup fail. Assume that the servers are ranked from 0 to k-1, starting with the primary server S_0. The failure of the first backup server S_1 will be detected by S_2. However, S_2 does not know that the primary has failed as well. Therefore, an election protocol is executed to determine the new primary. In this protocol each backup server S_i sends an rpc message to S_{i-1}. If no answer is obtained within some fixed time, another message is sent to S_{i-2}, and so on, until the primary S_0 is reached. If one of the backup servers does not get any answers, it wins the election and takes over as the new primary. Subsequently, the number of failed servers is determined, and missing servers are replaced.

In order to restart the neighbourhood surveillance protocol, the newly created servers must be included into the logical ring. The server which manages the reconfiguration of the service multicasts a message to all surveillance ports which contains the ranking numbers and surveillance port identifiers of all server replicas.

7. Related Work

Several techniques for achieving fault tolerance have been described in literature which are similar to ours. In the ISIS project [9] 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. A coordinator/cohort scheme [9, 10] 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 of the process group to join the cohorts. Therefore, the degree of fault tolerance decreases with each failure during a computation, making this approach feasible only for short-living applications. Unfortunately, ISIS provides only little support for making the coordinator/cohort scheme transparent for the application programmer who still is in charge of implementing an appropriate checkpointing function.

In the Delta-4 project several different fault tolerance techniques have been investigated. The passive replication technique described in [11] closely resembles our warm stand-by approach. Using this technique, 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 would render additional recovery actions unnecessary. In addition, the fault model adopted in

Delta-4 slightly differs from ours in that only node failures are considered while in our model server teams may fail individually. This fact enforces surveillance and failure recovery to be managed by the MOSKITO server teams.

For the Conic system [12] a hot standby scheme is proposed which is similar to our 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. As in Delta-4, only node crashes are detected by the hot standby managers, so that no explicit failure handling mechanisms need to be included into the application program code.

8. Conclusions

With the technique presented in this paper it is possible to provide non-fault tolerant application programs with mechanisms which allow them to be executed in a fault tolerant way. These mechanisms can be added automatically to the program code of a non-fault tolerant server, thus making the fault tolerance concepts completely transparent for the application programmer. In addition, the port group concept of MOSKITO efficiently hides the configuration of a service and its recovery after a failure from client applications.

Nondeterministic decisions taken by the primary are transferred to all backups by implicit checkpoints issued whenever the externally visible service state changes. This prevents the service from exhibiting an inconsistent behaviour in case that the primary fails and the backup re-executes a nondeterministic function.

References

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

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

[3] 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, 1990.*

[4] F. Cristian: “Agreeing on Who is Present and Who is Absent in a Synchronous Distributed Systems”, *Proc. 18th International Symposium on Fault-Tolerant Computing, 1988, pp. 206—211*.

[5] L. Lamport, R. Shostak, M. Pease: “The Byzantine Generals Problem”, *ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3,1982, pp. 382—401.*

[6] F. Cristian, H. Aghili, R. Strong, D. Dolev: “Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement”, *Proc. 15th International Symposium on Fault Tolerant Computing, 1985, pp. 200—206*.

[7] T. Becker: “Keeping Processes Under Surveillance”, *Proc. 10th International Symposium on Reliable Distributed Systems, 1991, pp. 198—205.*

[8] F.B. Schneider: “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial”, *ACM Computing Surveys, Vol. 22, No. 4, 1990, pp. 299-319.*

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

[10] 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.*

[11] 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, 1989, pp. 184—190.*

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