Ma 70

ACTA POLYTECHNICA SCANDINAVICA

MATHEMATICS AND COMPUTING IN ENGINEERING SERIES No. 70

Replicated Computations in a Distributed Switching Environment

RAIMO KANTOLA

Nokia Telecommunications
P.O.Box 33
FI-02601 Espoo, Finland

Dissertation for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Auditorium E at Helsinki University of Technology on the 9th of December 1994, at 12 o'clock noon.

HELSINKI 1994

Kantola, R., **Replicated Computations in a Distributed Switching Environment.** Acta Polytechnica Scandinavica, Mathematics and Computing in Engineering Series No. 70, Helsinki 1994, 144 pages. Published by the Finnish Academy of Technology. ISBN 951-666-430-X. ISSN 1237-2404.

Keywords: Fault-tolerance, replicated computations, migration of computations, switching.

ABSTRACT

Replication of computations in a distributed switching environment is studied. The first topics discussed are the requirements and the other design goals that have to be met by replicated computations in a distributed switching system. The requirements on the grade of service and availability performance objectives are largely set out in the international standards. A structured probability oriented software approach to building a kernel supporting replicated computations is suggested and the functional as well as the probability properties of the replication scheme are investigated. To aid the definition and investigation of the functional properties of the replication scheme a model of computation based on the actor model of Hewitt and Agha is defined and used. The overall replication scheme consists of a loose basic scheme, the real-time computation migration tools, here designated as warm-up algorithms, and the corrective replication tools augmenting the basic scheme. Language methods which enhance the transparency of the replication scheme are also discussed. The work has been done in connection with a redesign project of a distributed digital switching system and the results have largely been implemented in that environment.

Edition 2

14-Apr-14

ACKNOWLEDGEMENTS

The ideas concerning the work presented in this thesis started to take shape in the summer of 1987. About that time an ambitious specification project was launched to precede a major redesign of the processor system and the switching matrix on the hardware side as well as the basic software system of the DX 200 switching system. The DX 200 is a digital switching system designed, produced and commissioned by a Finnish company, Nokia Telecommunications.

A number of people from the company have contributed to the success of this work. The complete list of the people to be thanked for their help is too long to be presented here, but it includes many interested users, kernel implementors, tool experts and enthusiasts of switching software architecture. I am sure that we all remember the enthusiastic team oriented atmosphere of great fun without which any success in this work would hardly have been possible. Thank you all. However, I wish to mention some of the persons here: To Mr. Martti Syväniemi I am indebted for many stimulating discussions and ideas. Mr. K. Pasanen, Mr. Tero Peltola, Ms. Eeva Hartikainen, Mr. Matti Oosi and Mr. Ari Muittari are responsible for the current implementation, and especially Eeva has contributed to many details of the real-time computation warm-up algorithms. To Mr. Erkki Ruohtula I am grateful for his contribution in the language issues.

I also wish to thank the company and the management, especially Mr. Jussi Ilmarinen and Mr. Kai Kurru, for the opportunity and support needed to work out and implement ideas which arise from a background of working in different DX 200 design, specification and management positions mostly in the areas of various aspects of fault-tolerance and ISDN call control and signalling starting from 1981.

Coming from a practical engineering background a major effort for me has been the formalisation of the results. This would not have been possible without the guidance from Prof. Heikki Saikkonen and especially Dr. Markus Lindqvist. The idea of using an actor model to describe the semantics of computations comes from Markus who has also provided many valuable comments. Mikko Tiusanen PhD has provided invaluable criticism in finalising the text. Mr. Harvey Bensson has adviced on the use of English. I also thank Prof. Leo Ojala and Prof. Sulonen for their support.

Finally I want to thank my wife Natalia for endurance and motivation to finalise this work in the form of this thesis.

Helsinki, October 1994,

Raimo Kantola

This work is dedicated to the Order of DX Knighthood and in particular to the Chastity Committee of Kalkki.

BACKGROUND

The history of the DX 200 system goes back to the emergence of microprocessor technology in the early 1970's. Initially Intel's 8-bit microprocessors were used as the main processors and the first fully digital local exchange in Europe was put into commercial service in 1982. Later, until 1990, the main processors were based on Intel´s 8086 and 80286 architectures. The system has been developed in incremental stages called system releases. Each new release has increased the functionality and applicability of the system to public telecommunication networks. So far, over four million equivalent subscriber lines of the DX 200 technology have been commissioned in a number of countries.

The system redesign project initiated in 1987 intended to take an advantage of the most recent developments in microprocessor and VLSI-technology with the specific goals of

- upgrading the system capacity at least fourfold i.e. up to 400 000 BHCA and 10 000 erlangs, and creating a potential for further increase in capacity,
- preserving the possibility of upgrading all the existing DX 200 systems to the new system release level,
- removing all the software architecture limitations and thus,
- creating a comprehensive core system for a number of new telecom applications, like the Intelligent Network systems, Centrex, GSM mobile exchanges and other GSM network elements, concentrators, etc,
- designing those applications, and
- reducing the cost of ownership, by integrating software and data package update support features into the exchange system software and by wider use of random access memory for storing the software as well as data.

The first public exchange delivery of the new system level took place in March 1990 and the exchange was taken into commercial use in August 1990 as planned.

Intel 80386/486 processors[1] were used to implement all the control computers. During the implementation phase it turned out that with the 386 control processors, call processing capacities of 600 000 BHCA could be achieved, while with plug-in compatible high end 486 control processors, call processing capacity of over 1 000 000 BHCA has been demonstrated.

The architectural development efforts involved more specifically:

- new operating system kernel,
- embedding replicated computations support into the kernel,
- redesign of the memory resident file system,
- new memory resident distributed database system,
- new call processing architecture, and
- separating program interfaces from the implementation.

This thesis concentrates on the replicated computations support problems. The ideas for this thesis were worked out or at least inspired and largely implemented by the project described

[1] 8086, 80286, 80386, and 80486 are Intel microprocessor products.

above. My role was to be the initiator and the head of a task force working on replicated computations as well as heading the System Maintenance software development team.

Contents

GLOSSARY OF SYMBOLS AND ABBREVIATIONS

Symbols

$=$	equals,
$\neq$	is distinct from,
$\cong$	approximately equals,
$\simeq$	is isomorphic to,
$\in$	is a member of; e.g. $a \in A$: a is a member of set A,
$\subseteq$	is a subset of ; e.g. $\mathrm{P} \subseteq \Pi$: P is a subset of Π.,
$\bigcup$	a union of sets,
$\bigcap$	intersection of sets,
$-$	complementation of sets, e.g. A – B is the complement of B relative to A,
Æ	an empty set,
$f\vert$	a function restricted to a sub domain,
$\Rightarrow$	implies,
$\neg$	not,
$\forall$	all, for all,
$\exists$	exists,
$\exists!$	exists as unique,
$\mathrm{O}(x)$	grows as expression x; e.g. $\mathrm{O}(n)$ grows linearly as a function of n,
$<<$	substantially smaller than,

Abbreviations for functional unit states

WO	working, synonym active,
SP	spare,
SP-EX	spare-executing, i.e. hot-standby state for $2N$ replicated units, ready for binding to form a temporary pair for N+1 redundant units,
SP-UP	spare-updating, an intermediate unit state for unit warm-up before it can become SP-EX,
TE	testing, a unit state dedicated to fault location,
BL	blocked, a unit state in which no new tasks are allocated to the unit,
SE	separated, a unit state for repair actions,

Other Abbreviations

BHCA	Busy Hour Call Attempts, a measure of performance of a switching system,
CCITT	Comite,´ Consultatif International Telegraphique et Telephonique. CCITT has recently renamed itself as Telecommunication Standardisation Bureau of the International Telecommunication Union,
CCS7	Common Channel Signalling System # 7,
CPU	Central Processor Unit,
DX 200	a switching system developed and manufactured by Nokia Telecommunications,
DMX	the real-time Operating System of the DX 200 Switching System,
FIFO	first-in-first-out queue,
GSM	Originally Groupe Spe,´cial Mobile, now Global System for Mobility, Digital Mobile Network standard,
iff	if and only if,
ISDN	Integrated Services Digital Network,
I/O	Input/output,
kbyte/s	thousands of bytes per second,
Mbyte/s	millions of bytes per second,
MTBF	Mean time between failures,
MTP	Message Transfer Part, lower layers of CCS7 specified in [Q.7XX],
MTTF	Mean time to failure,
2*N*	a redundancy principle which assumes that *N* active functional units share the computing load created by an application and that each of these units has a spare permanently bound to it,
N+1	a redundancy principle which assumes *N* active units share the computing load created by an application and that there is one additional unit to recover from faults,
O&M	Operation and Maintenance,
OMU	Operation and Maintenance Unit,
OSI	Open Systems Interconnection, a model of communications between open systems specified by the International Standardisation Organisation,
SDL	Specification and Description Language (defined by the CCITT),
SIG	Signalling and Call Control Unit,
SU	Service Unit.

PREFACE

Fault-tolerance is the property of a computing system to perform in a satisfactory fashion in the presence of faults. Faults are deviations from the intended behaviour of the system [Re84]. Ideally all computing systems should be fully fault-tolerant, but because of economic considerations and because of some theoretical limitations, this is not so in practice.

The reliability of the computing system can be increased by adding redundant hardware. Acceptable cost of redundant hardware varies in different application areas and depends on the importance of fault-tolerance for the application. When human lives or the entire mission of the system are at stake, a significant increase in the cost of the whole system because of redundancy is usually acceptable. In switching, faults are more of a nuisance and a cost factor than anything else. Faults in switching systems cause degradation of service provided by the telecommunication network, but customers do accept a certain level of erroneous behaviour of the network. A crash of a switching system causes the operator to loose revenues from the customer connections and in a competitive environment both the switching system vendor and the operator get bad publicity which does not help in gaining new business. The maintenance costs of the systems also have to be taken into consideration.

Therefore, in switching system design the costs of increasing the reliability of the system have to be compared and balanced with the economic losses of the operator and the vendor due to failures of the systems. At the same time an acceptable grade of service, as defined by international standardisation bodies like the CCITT and the network operators, has to be maintained. For switching systems, an acceptable and sufficient level of redundancy seems to be duplication or even a lower level of redundancy, as in our example, the DX 200 system. In such systems normally some fault detection time is required before recovery actions can be started. Under these conditions it is not possible to mask even all single faults. For this to be possible, at least three concurrent computations or two self-checked computations executed by different computers are needed [Re84]. Switching systems are manufactured in large numbers, which is why − although massive hardware redundancy has to be ruled out − significant design efforts are quite acceptable to come up with a good solution to problems in the area of fault-tolerance.

In this thesis, we will study more thoroughly one of the problems in the design for fault-tolerance, the organisation of *replicated computations*, under the external requirements prevailing in a switching environment. The task can be formulated briefly as follows: we have to suggest an organisation of replicated computations which would facilitate fast implementation of new replicated applications and which does not bear significant cost penalties in the control part of the targeted distributed switching system. This also excludes solutions with a high performance penalty. A high performance solution is not important as such but rather because poor performance costs money.

This study has been inspired by a redesign project of the DX 200 switching system starting from 1987 and finished by 1991. The results of this study have been largely, although not totally, implemented in the DX 200 system and in the tools used in the design process of that system. The run-time replication tools were made available to the applications in 1991 and have since been taken into commercial use by an increasing number of applications. The implementation of the tools and their use in a large number of applications has now been debugged and the tools are being further developed based on this experience. The tools have been rather popular among application designers and experience shows that the target of fast implementation of new replicated applications has been achieved although the debugging of such applications constitutes a considerable effort. An example of applications using these run-time and the corresponding design tools are Nokia's GSM network elements based on the DX 200. Where bigger suppliers had considerable difficulties Nokia succeeded in the timely introduction of the whole range of GSM network elements partially due to the language and replication tools which will be the focus of this study.

In all Chapters of this thesis we have attempted to structure the material into general and DX 200 specific where applicable, presenting first the general concepts and then specifics of the DX 200 system separately. We have adopted the terminology from the DX 200 vocabulary as such where the terms are well established. In some cases we have, however, deviated from the current DX 200 term in favour of a more generally understandable word.

Our presentation in Chapter 1 will start with a description of the computing environment. In particular we shall discuss distributed switching systems in general and review those features of the DX 200 system which are relevant to the study of replicated computations. In Chapter 2 we will look more thoroughly into the requirements to be met by our design and formulate our design problem more precisely.

Chapter 1, together with the requirements analysis in Chapter 2, forms the basis of *requirements traceability*, i.e. that we have correctly understood the basic needs and the consequent realistic assumptions which we have to bear in mind while solving the fault-tolerance problems and how those needs and assumptions are transformed into features of the solution. This concept is fundamental and intended to direct any abstractions that we may wish to adopt in such a way as to keep us strictly within practical engineering problems.

Chapter 3 covers different approaches in fault-tolerance design and especially in the area of replicated computations. This study shows that there is a demand for a new design in the area of replicated computations, which would be well suited for the needs of the applications in a distributed switching system. The suggested approach is structured and is presented as a set of replication tools provided by the kernel software to the applications.

Chapter 4 presents the basic modelling tools which will be enhanced and used in the following chapters to formulate the properties of the replication scheme suggested in this thesis. Our model of computations is based on the actor model suggested by Gul Agha [Ag, Ag86] and used in [Lin91]. The model does not limit itself to message passing as the sole means to communication between processes; the existence of a memory resident file system is modelled by *data units* accessible by several processes. Chapter 5 first formulates a model of replicated computations and then lays out the basic replication scheme on top of which a set of other replication tools are developed. The intention of a replication scheme is to implement requirements of the model of replicated computations. To our knowledge the suggested replication scheme is a unique development of the approach based on *eventual convergence*, i.e. the scheme does not aim at instantaneous correctness at the basic level. Due to this fact, as we shall see in Chapter 10, high performance is achieved in the type of applications which are typical in switching.

One of the specific replication problems, namely real-time migration of computations or computation warm-up, as we have called it, from a computer unit to another is formulated and a satisfactory solution is suggested in Chapter 6. The purpose of the suggested algorithms is to allow for a graceful transfer of call control, signalling and statistic computations from a computer unit to another in case of failure of the active unit or by an operator command. The warm-up algorithms do not use roll back, i.e. are not based on re-executing any code by the new active unit, and thus are well suited to real-time environments. Such real-time algorithms for processes communicating by message passing and also using memory resident files have not been presented before.

In Chapter 7 the results of the previous Chapters are used to define the sufficient conditions for corrective actions in the case of an replication error in the spare computation as well as to define more exactly the concept of *conditional instantaneous correctness*. These theoretical results show the possibility and the goal of the additional replication tools that aim at a higher level of correctness of the spare computation. The best of the tools, presented in Chapter 7, can be applied selectively to important events by the programmer and thus do not destroy the performance characteristics of the basic replication scheme. Markov's reliability modelling techniques are used in Chapter 8 to study the probabilistic properties of the suggested replication scheme. The reliability analysis makes it possible to compare different corrective replication tools. Chapter 9 addresses the problem of transparency of the replication scheme. Language tools which allow the use of the replication primitives of the run-time system in a declarative fashion are described. These language tools increase the level of transparency of the replication scheme.

Chapter 10 evaluates some properties of the suggested approach to replicated computations and lays out the contribution of this thesis to the knowledge on fault-tolerance techniques.

1 Introduction

In this chapter we will first describe an abstract structure of a switching system and especially our view on the computer architecture of the switching system. Next, we will discuss such aspects of our abstract switching system as fault-tolerance and its implementation by software methods and hardware redundancy. We will then look at the computing environment we assume to prevail in our switching system. To conclude our introduction we will present some specific characteristics of the DX 200 architecture in Section 1.5.

1.1 What is a switching system?

Public switched telecommunication networks are mainly composed of transmission systems and switching systems. From the point of view of the end-customer, transmission systems provide connectivity on layer one, the physical layer, of the Open Systems Interconnection model while switching systems work on layers two, the link layer, and three, the network layer. Consequently, switching systems provide dial-up connections.

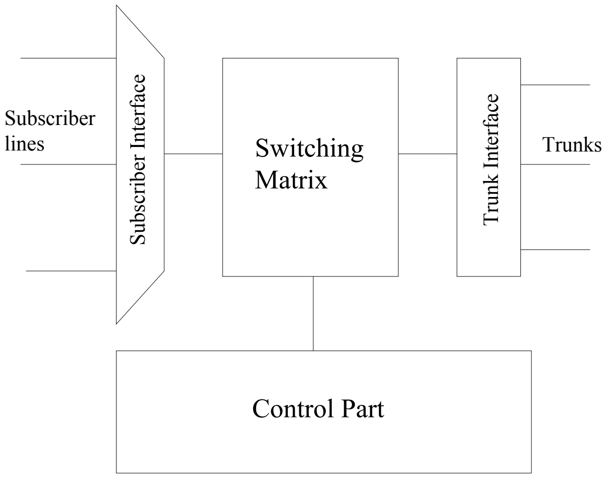


Figure 1.1. An abstract switching system.

In a switching system architecture four principal components are needed (see Figure 1.1): the *subscriber* interface, the *trunk* interface, the *switching matrix* and the *control* part. The subscriber lines are connected to the subscriber interface and the trunks to other exchanges of the network are connected to the trunk interface. Subscriber lines may be either copper pairs or 2Mbit/s connections. Trunks are 64 kbit/s connections carried in time-slots of 2Mbit/s signals. The term 'trunk' refers to the fact that these connections, as opposed to subscriber lines, are used to carry calls to and from *any* of the subscribers of the system, the mapping being established at the call set-up. The switching matrix allows the establishment of circuit switched connections on a call-by-call basis at least on the 64 kbit/s level.

The control part is connected with the network environment through the subscriber and the trunk interface, and with the switching matrix. From the point of view of the control part, the switching matrix is just a resource that has to be managed.

The control part is typically a distributed computing system running the control software. The current public switched telecommunication networks have been built during several decades and by using equipment from a large number of vendors. Consequently, it is not surprising that a large number of different types of protocols for both the subscriber and the trunk side as well as passing call control information, i.e. *interworking*, between all of them have to be supported by the control part of the switching system. None the less important is the requirement that accurate information on all calls has to be collected e.g. for charging purposes. To be able to provide service, the control part has to incorporate a lot of data including the subscriber database and a description of the surrounding network.

Sometimes we will also use the term *exchange* as a synonym of a switching system.

1.2 Computer architecture

A switching system has some application oriented hardware but above all it has a powerful computing system and a lot of software. In our experience, some 80 to 90 per cent of the continuous design effort goes into software development. Different switching systems have adopted different computer architectures. Some systems are centralised, some are distributed. Among distributed designs, both shared memory systems and multicomputers, i.e. processors with their own memory have been used. Various communication mediums and structures between the computer units in distributed switching systems have been used as well. These include message switches, communication through the main switching matrix and bus structures.

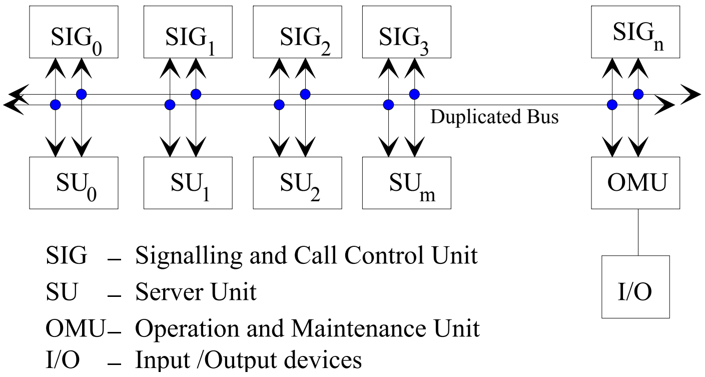

SIG$_0$
SIG$_1$
SIG$_2$
SIG$_3$
SIG$_n$
Duplicated Bus
SU$_0$
SU$_1$
SU$_2$
SU$_m$
OMU
I/O
SIG – Signalling and Call Control Unit
SU – Server Unit
OMU– Operation and Maintenance Unit
I/O – Input /Output devices

Figure 1.2. The Computing system.

We will assume that the computer system is composed of *computer units* and a communication structure. For short, we will often use the terms unit and computer as synonyms of a computer unit. The computer units are based on processors with their own local memory and there is *no shared memory*. We will assume that the communication structure between the processor units provides *full connectivity*, i.e. any processor can send messages to any other computer, and that it is not the worst performance bottleneck in the system. In our experience to achieve the call processing capacity of more than a million BHCA with commercially available processors (e.g. 80486) this requires message passing delays of less than one millisecond in the communication medium and a bandwidth of at least on the order of 8 Mbit/s. We will also assume that the *order of messages does not change* in the communication medium and that access delays to the communication medium are predictable if the communication medium is shared. In practice such a communication medium can be e.g. a fast bus. Incidentally, such a multicomputer structure is similar to a network of workstations and servers connected by a local area network.

We will also assume that at least one of the computers has access to non-volatile memory and to input/output devices. Such a computer is necessary to perform a system restart and to provide a man-machine interface to the user (operator) of the system. Figure 1.2 presents such a multicomputer system. In Figure 1.2 we have adopted a duplicated bus as the communication medium although our reasoning is not limited to the system in which the communication structure is a bus. We have named the computer with direct access to the I/O, the *Operation and Maintenance Unit* (OMU). We have also assumed that not necessarily all the computers are identical. In Figure 1.2 a server unit is a centralised computer providing a certain set of centralised services e.g. the subscriber data base or charging and performance data collection and management. The signalling units communicate with the network environment i.e. the subscribers and other switching systems performing the appropriate communication protocols, interworking and call control tasks. Our approach does *not* presume, however, that such a clear functional distribution between the server units and the signalling units is adopted.

An important feature of our assumed computing system is that it is scalable and reconfigurable. Scalability means that the system can be expanded by adding new processor units in step with the growth of processing needs. Dynamic reconfiguration of the existing computers is used as a means to recover from faults.

1.3 Fault Tolerance and System Maintenance

One of the functions performed by the control part of a switching system is system maintenance. Its purpose is to handle all the fault situations and user initiated configuration management tasks in the system hardware and software in such a way that all the *availability performance* requirements for the whole system and for the individual customers are met. The availability performance issues will be further discussed in Chapter 2.

Fault tolerance is a general requirement for all the software of a switching system, but above all system maintenance is responsible for this system feature. The structure of system maintenance is illustrated in Figure 1.3.

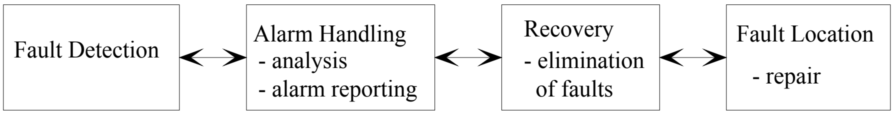


Figure 1.3. The Structure of System Maintenance.

System maintenance contains functions for hardware and software *fault detection*, *alarm handling* for analysing the fault information from different sources and informing the operator, *recovery* for eliminating the effects of faults, and *fault location* to aid the operator with the fault repair activities.

For maintenance purposes the system consists of *functional units*. Faults are pinpointed into the functional units by the alarm system and replication of hardware is handled by the recovery on the level of functional units. Physically functional units consist of replaceable plug-in units to which faults are pin-pointed by the fault location function. Repairing the system means changing the faulty plug-in unit.

Functional units have *operating states* and *substates* managed by the recovery. For simplicity we will mostly adopt the DX 200 terminology for these operating states. The states are *active* or *working* (WO), *spare* (SP), *blocked* (BL), *testing* (TE), *separated* (SE). Active units handle the call traffic and run the other active application functions. SP units are the stand-by units ready to take-over in case of a WO unit failure. When a unit fails, the recovery takes it to the TE state and starts the fault location function. Before fault location, some units may be blocked (BL) to stop the allocation of new tasks to them. Fault location runs all the diagnostics programs for the unit and locates the fault into a replaceable plug-in unit or gives a list of suspected plug-in units placed in the estimated probability order. The unit is taken to the SE state for plug-in unit replacement by the operator. After replacement the operator verifies the results by starting the diagnostics programs with an operator command.

The functional unit state is an important concept and enables the recovery system to control all the applications running in one unit as a whole. In our view this is different from what seems to be a common approach e.g. in transaction processing systems in which the basic model of computation includes a recovery action, namely, abort. The functional unit concept is one of the factors supporting the idea of a hierarchical recovery system instead of dealing with all the recovery problems on the process level. The recovery from faults is considered a difficult task and − as is most often the case − hard problems are tackled with a hierarchical approach. The recovery subsystem is a three (in some cases four) level hierarchy, with components in all pre-processors, in the control computers and with the central recovery control function, which is, under normal conditions, located in the O&M unit, but may be dynamically relocated into a pair of computers e.g. in the case of an O&M unit failure.

Fault detection may be based on software as well as hardware and all the applications should inform the system maintenance about all the errors in their environment and about internal inconsistencies. The design approach for applications should include being prepared for practically any conceivable errors in their environment and to report those errors to the system maintenance.

1.3.1 Main Hardware Replication Schemes

We will assume that at least two replication schemes are used in the switching system. In both of them *N* units are needed for tasks of a certain type, where *N* is a small integer for a system which the operator can change by installing new equipment and activating it by operator commands. The schemes are:

2*N*

Two units execute the same task and one of the units is always active or in the working state (WO) and the other unit is kept in the hot stand-by state or the spare (SP) state.

Replaceable *N*+1

There is just one or a few spare units and these are not used by the applications and are not permanently bound to any of the *N* active units but can take over the load of any one of them in case of an active unit failure or by an operator command.

When a spare unit is bound to an active unit by a changeover command, a pair is made up, the spare unit is *warmed-up* to the hot stand-by state, i.e. the computational state of the active unit is cloned to the spare when the spare is taken from the TE state to the SP state, and the command initiated changeover can be completed without major service interruptions.

A common state diagram can be adopted for both types of replication schemes, shown in Figure 1.4. Computer units using any of the two replication schemes can be managed by the recovery system by implementing the common state diagram. The state changes in Figure 1.4 are (1) *unit changeover*, i.e. the active unit becomes spare and the spare becomes active, and (2) the spare unit is handed over to the diagnostics system for fault location or returned to the spare state after verification.

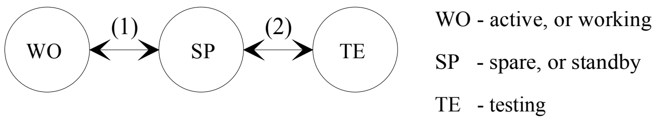


Figure 1.4. Basic state diagram of a computer unit.

1.4 Computing environment

The computing environment consists of the *programming model* defining the program objects that the programmer has to deal with and the common *data services* available to the programmer. We will deal with each of them in turn.

1.4.1 The programming model

This model is needed to capture the programmer's view of the system and to target the issue of application transparency.

The programming model is based on the message passing paradigm and the concepts of extended finite state automata and communicating asynchronous processes. This model inherits the message passing, state automata and process concepts from SDL [Z.100]. This basic paradigm is augmented with the concept of a *service* [Lin91, TNS91] to facilitate the description of the external behaviour of an object[2] without the knowledge of its internal structure. This programming model is supported by the TNSDL language [TNG, TNS91] which implements a version of SDL, augmented with a model of the system and the concept of a service.

[2] Objects may be processes, blocks, and asynchronous or synchronous procedures as defined in SDL.

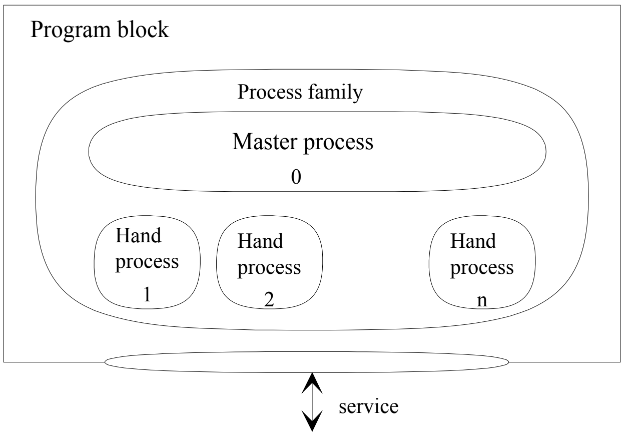


Figure 1.5 Main program objects.

Figure 1.5 presents the main static and dynamic program objects. The static code objects, the *program blocks*, have an *interface* and an *implementation*. A program block is a kind of load module containing the executable code of a *process family* or a library. A process family contains at least one process, called the *master process*. Many process families may be created on a program block code which is not a library. Creating the process family, in the kernel sense, means creating the master process. A family may also contain a number of *hand process*[3] *types*. The master may create and allocate *hand processes* for distinct computations, which are instances of a hand process type. Depending on the nature of the computation, a hand process may be permanently allocated to a task and always ready to function, or it may be allocated for a certain period. The time initially requested may be prolonged with a new request. The idea of the model is that a master process is simple and mainly allocates hand processes for parallel computations and the *hands do the actual work*. Additionally the master process may have other house-keeping tasks such as dealing with messages addressed to hands that do not exist.

The requirements of modularity, ease of software maintenance, and ease of re-use lead to a desire to decompose the signalling, call control, and other telecommunication applications into a large number of processes. This implies that several hand processes may be allocated to a call, each handling just a compact modular function, like incoming network layer signalling, incoming call control, resource handling, outgoing call control, outgoing signalling etc.

A typical process has an initialisation routine after which it waits for a message or messages to arrive. A typical process has only one point in code called the *main receive* point where it waits for messages to arrive and where it always returns after having processed a message. Processing a message may include changing the values of the *state variables* of the process and sending some messages. Always, when a message e.g. an acknowledgement is expected to arrive within a time period, a time limit is set. This ensures that the process is able to recover in case of an error.

A process also has the *kernel state variables* which are set by the system rather than by the application. The kernel state variables include scheduling state and the communication state. The communication state of a process may be "reachable", i.e. incoming messages are normally delivered to the process or "unreachable", i.e. incoming messages are discarded or e.g. they may be delivered to the master process of the unreachable hand. We will assume that messages are always received in the same order as they were sent by a single process. Two messages sent by two different processes to the same destination may be received in a different order than they were sent.

[3] We live in a democratic country but work hard, so our processes cannot be slaves nor children.

computer_address	family_id	local_process_id	focus

Figure 1.6. The Process Identifier.

A very important feature of any programming model of a distributed system is how messages are addressed to their destinations. Here we shall present the basic solution and add details later when they are needed. Each process instance in the system has a *process identifier* or *Pid* (see Figure 1.6). From the point of view of the programmer, the Pids are unique. The Pid is used to address the messages to the recipients. The Pid is composed of the *computer address*, the *family identifier*, the *local process identifier*, and finally the *focus* differentiating the subsequent lives of the process instance. The allocation of identities to computers and families is static. However, computer address allocation is static only from the point of view of the applications, while the recovery control may remap the computer addresses to the physical computer units in a unit changeover. The local process identifier and focus allocation is dynamic. The local process identifier of the master process in a family is always zero.

We can see that this addressing method does not support complete *location transparency*. This is because a computer address is a structural part of the Pid. Complete transparency would mean that the Pid should not have any structure but the operating system should map the Pids directly to the processes of the system. It seems that complete location transparency in a distributed system like ours with a very large number of processes would require either very large addressing tables or elaborate kernel features for optimising such tables. The question is: do we need complete location transparency? The benefit would be that arbitrary process migration could be implemented transparently to the applications. It seems that in an embedded switching system environment in which the task of the system is always the same, the need for this level of dynamic behaviour of the system is not obvious. That is why the described simple addressing mechanism allowing for effective implementation and for hiding the unit changeovers from the applications seems to meet the switching system requirements and was also chosen for the DX 200 system.

In a computation the initiator is called the *service user*, and the process requested to execute a function is called the *service provider*. The identity of the service provider may not be known. Then the corresponding master is requested to allocate a hand for the task. So, the initial message to request a service may always be sent using the static address of the master process. The possible subsequent messages of the service may be sent using dynamic addresses of a hand process established at run-time. The identity of the service user is given to the provider hand in the allocation, and the identity of the service provider is given to the user in the acknowledgement which is usually sent by the service provider hand.

Our model differs from the client-server model in that asynchronous communication is employed. We will not seek to describe services formally, although a suitable notation exists [TNG, TNS91, Lin91] but will only adopt the service user - provider terminology for describing relationships between objects in our environment. In an asynchronous communication environment the roles of the user and the provider can be attached to processes only in connection with a certain service.

When we talk about creating a process, we mean creating an instance of a process type. To avoid the cumbersome `process instance´ we will in most cases use the word `process´ instead. However, in formal presentation when the meaning is not obvious from the context we will resort to the term `process instance´.

It should be noted that when a programmer defines a message, he defines a data type. When the program is executed, a message is an instance of the corresponding data type. Reception or sending of a message instance is called an *event*. For short and to avoid repetition we will often use the word message instead of message instance.

This powerful programming model has shown its worth in the application design for the DX 200 system. The design methodology and tools based on the model seem to increase the productivity of a designer considerably, compared to using only a language like C, although statistical data supporting this claim has not yet been collected [Sea91].

1.4.2 Memory resident file and database systems

In a switching system a large amount of exchange specific data is needed e.g. for describing the network environment and usage, the subscriber service features, charging and different measurement arrangements and the exchange itself. At the same time the exchange collects a wide variety of different types of data about the process being controlled by the system. For performance reasons most of the data has to be memory resident to give the signalling, call control and charging functions a fast access to this data, and it is for this reason that a memory resident file system is necessary in a switching system.

The memory resident file system is most efficient if the files are accessed by the applications without buffering by means of a set of memory resident library routines. Consequently, they form a kind of extension to the data areas allocated for the variables of the applications.

Although message passing is the main way of passing information between processes, the existence of the memory resident file system means that data sharing through such files have to be considered when devising algorithms for computation replication schemes. These problems have not been considered before in the literature, e.g. in [Co85], [Bi85], or [Ch88].

Such features of transactions as the atomicity, serializability[4] and permanence of data changes are not tackled by the memory resident file system alone. These features can be partially embedded in the operator command application programs, the file update system, the man-machine interface and input-output systems. Alternatively, these important features of a switching system can be implemented in a memory resident database system. The database system should guarantee atomicity, serializability, and at least partial permanence of transactions, be well suited to the distributed computer system architecture and have very high performance. Permanence of the results of a transaction mean that they can not be undone. Partial permanence means that transaction results are not updated to non-volatile memory at the transaction commit time because of performance reasons. At commit time the results have to be updated at least in two computer units and they are transferred to the disk asynchronously on the background. Such a database system has been implemented e.g. in the DX 200 system and is being used by a number of applications.

1.5 The DX 200 system architecture

The control part of the DX 200 system is a loosely coupled microcomputer network. The processors have no common memory and are based on Intel´s 80386/486 microprocessors. The control computers are connected with a proprietary fast parallel synchronous message bus. Each of the control computers may have pre-processors at two levels e.g. for real-time signal processing. Pre-processors are point-to-point connected with their particular control computers and are managed using the master-slave scheme. The bulk of the software, which comprises about five million lines of code, is concentrated in the control computers which are the focus of our attention also in this thesis.

[4] For further discussion on atomicity and serializability, please see Section 3.3.

The DX 200 system and its computer subsystem are functionally distributed. This distribution is implemented by the concept of *functional unit types*. A functional unit is an entity of hardware or software and hardware, dedicated to a function or a task. The main attributes of the functional unit abstraction are:

- the functional unit type,
- the functional unit index,
- the unit states and substates, and
- the addresses.

The functional unit type and index are used for unit identification throughout the system. The overall functioning of the system is controlled by controlling the states of the functional units of which the system is composed. The most important of the functional units are the control computer units. The computer units have addresses, which are used for communication between the units.

The unit typing is generic in nature: a feature classification scheme is used to implement the unit type concept. The result is that new dedicated unit types required by new applications are easily incorporated into the system architecture.

DX 200 is a loosely coupled distributed, fault-tolerant system with some centralised functions. The DX 200 system maintenance functions conform to the idea of loose coupling of autonomous control computers. This means e.g. that

- access to state control mechanisms does not depend on any centralised hardware,
- control computers control their own software and are able to restart on their own, requesting the appropriate boot and loading services as needed,
- the control computer states are controlled from a centralised point which may, however, be dynamically reallocated from the OMU to a pair of control computers e.g. in case of an OMU failure,
- only the control computers are autonomous as described above, the pre-processors run under the control of the master computer unit.

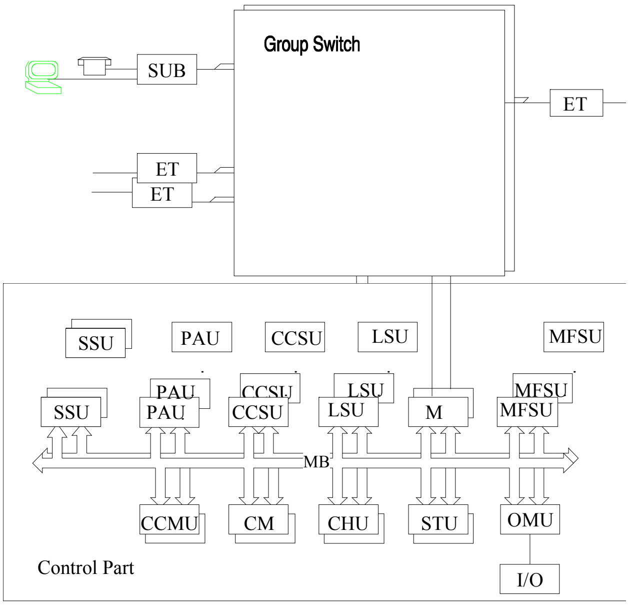
In line with these principles the following design objectives for the hierarchical recovery system are set:

- an individual program block failure or a single hardware failure should have a minor effect on the overall functioning of the whole system,
- very infrequently the fault situation should be so bad or complex that the whole system has to be restarted. Most often, restarting a program block, a pre-processor or a single control computer should be enough.

The achieved level of system availability confirmed by field experience reported e.g. in [PuAf] and customer comments support our confidence that these objectives are met by the actual design.

1.5.1 Examples of DX 200 systems

The hardware architecture of *fixed network* exchanges of the DX 200 system are shown in Figures 1.7 and 1.8. These figures should be understood as examples of possible DX 200 systems. Another application, e.g. a DX 200 Mobile Switching Centre for the GSM network, would have a different set of application specific computer unit types.



SUB	– Subscriber Module
ET	– Exchange Terminal
SSU	– Subscriber Signalling Unit
PAU	– Primary Access Unit
CCSU	– Common Channel Signalling Unit
LSU	– Line Signalling Unit
M	– Marker Unit
MFSU	– Multi-Frequency Service Unit
CCMU	– CCS7 Management Unit
CM	– Central Memory
CHU	– Charging Unit
STU	– Statistics Unit
MB	– Message Bus
I/O	– Input/Output
OMU	– Operation and Maintenance Unit

Figure 1.7. DX 200 Hardware Architecture for a large exchange configuration.

The control part (see Figure 1.7) consists of functional computer units for different types of signalling and call control (SIG) as well as service functions (SU). A group of SIG units is allocated for each of these main types of signalling. Computers in these groups are said to be of a specific unit type and the unit type has a name. The SIG units in a fixed network DX 200 exchange take care of such functions as analogue and digital subscriber signalling and services (SSU), primary access signalling and call control (PAU), common channel (CCSU), and channel associated (LSU) signalling, etc. All computers of a specific unit type run identical software. The system can be extended by creating new SIG units on-line as additional external lines are connected to the system and more call processing capacity is required. In a switching system this is necessary because expansion of the system is *not* a good enough reason for planned system down-time.

In a given installation there may be service units such as the Statistical Unit (STU), the Charging Unit (CHU), the Central Memory (CM) or the data base unit, etc. When more processing capacity is required, additional units may be created by splitting the service unit functionality or adding new specific functional unit types to the system. Two dedicated service units, namely, the Marker (M) for switching matrix control and the Operation and Maintenance Unit (OMU or O&M unit), are also part of the control system. The SUB is a dedicated functional unit type to allow connecting subscribers to the system. The ET is an exchan-ge terminal, a functional unit type supporting one 2Mbit/s connection to the system.

Let us look at an example of a set-up of an originating outgoing call from a local subscriber and see what functions are performed by different types of computer units. As the call originates from a subscriber physically connected to the system, the *off-hook* condition is first recognised by the subscriber module. The SUB performs analogue to digital conversion and adapts the signalling on the subscriber line to the internal signalling on a point-to-point signalling link through the switching matrix to an appropriate pre-processor of the subscriber signalling unit, SSU, allocated to that subscriber line. The SSU takes care of the network layer signalling and incoming call control, generates the charging information, and manages the circuits for the call. The incoming call control requests subscriber data and number analysis services from the Central Memory and passes charging and performance data to the charging and statistics units. The incoming call control further requests the Marker, M to seize the necessary circuits and make the connections e.g. to the auxiliary equipment such as tone generators and announcement machines. At some point during the analysis it finds out the identity of the outgoing circuit and consequently the identity of the outgoing call control computer. In our case, we will assume that this is one of the common channel signalling units, CCSU. The CCSU performs outgoing call control and outgoing network layer signalling. The CCSU also performs the distributed signal routing functions of the message transfer part of the CCS7 signalling [Q.7XX]. The CCS7 link layer protocol is performed by a pre-processor of the CCSU. The Common Channel Management unit is not involved directly in call handling but controls e.g. CCS7 link state changes and link sets. When the called party answers, this condition is recognised by the outgoing call control and the information is passed to the incoming call control. The incoming call control requests the Marker to make the through-connection of the circuit between the calling and called parties and initiates charging of the calling party. This completes the set-up phase of the call.

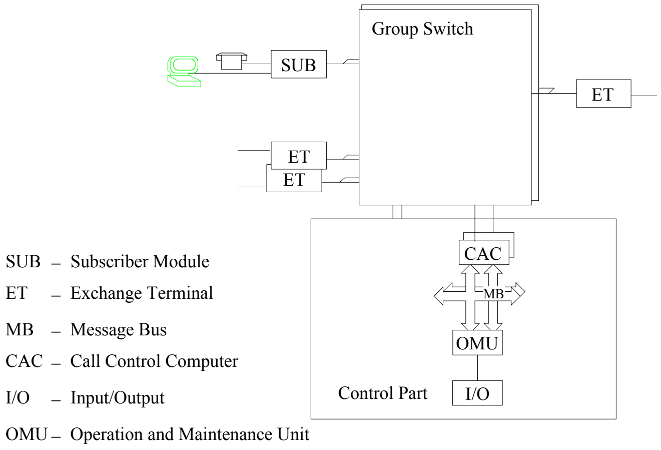


Figure 1.8 DX 200 Hardware Architecture of a small exchange.

As well as extended, the control system can be shrunk into e.g. three control computers. This has been done in the small exchange configuration, presented in Figure 1.8 and called the DX 210. In this configuration the Call Control Computer takes care of all functions of all the signalling units (SIG) and service units (SU) of the previous configuration.

The trend for the need of different functional unit types in public switches is driven by several factors. One is the continuing growth of the level of hardware integration. An example of this is that new signal processing chips are widely applicable for all kinds of channel associated, as well as message based signalling commonly used in public telecommunication networks. A conflicting factor is that lots of different network element types are emerging in the public telecommunication field, and to handle these new functional unit types will be needed in such a system architecture. This suits the architecture well, because new dedicated functional unit types can be created by setting new values to parameter tables, if the new unit type falls within the present internal classification scheme.

1.5.2 Real-time operating system

An example of how the programming model of Section 1.4.1 can be implemented is the real time operating system kernel of the DX 200 computer system. The kernel scheduling the DX 200 control computers and supporting asynchronous message passing is called DMX. The kernel supports a kind of location transparency, because the message structures and primitives are the same for messages inside a computer as well as between the control computers.

The kernel supports fault-tolerance in many ways, one of which is the *memory segmentation scheme*. In this thesis we will, however, be mainly interested in the computer and process replication support by the operating system kernel. We will not go into the details of kernel implementation, but rather describe the computing, and replication models and the replication scheme which lay behind the kernel implementation. About the implementation of the kernel we will go only as far as to clarify that here we have used and will continue to use the term kernel meaning both the operating system kernel itself, the kernel support library of the implementation as well as a number of kernel processes, some of which will be discussed in this thesis.

The DMX uses memory *segments* for *memory protection* and addressing. Each segment has a *selector* containing the start address, the length and the access rights. For example, memory is allocated dynamically for program blocks and the memory resident files during loading or creation time. Consequently, e.g. the files may be located in different absolute memory addresses in a pair of WO and SP computers. Each file is placed in a memory segment of its own. The files are always accessed through the *file handle*, which contains e.g. the selector of the segment. This scheme helps to control the access rights to the files and gives the possibility to use memory protection for files.

With the `create_hand` kernel primitive the master process can create a large number of *hand processes.* A process may be *a heavy process* or *a lightweight process*. A process family is composed either of heavy or lightweight processes. Each process has its own block of control data, called the *control block*, used by the kernel containing e.g. the *kernel state* variables of the process, and its own *data segment*. The *code segment* is common for a program block. A heavy process has its own *stack segment* and its own *queue of incoming messages*. This means that heavy processes may be scheduled completely independently of one another. All the lightweight processes of the program block have a common stack and a common queue. This means that they use much less memory than heavy processes and a control computer is able to run even tens of thousands of lightweight processes. Consequently, in a lightweight process there should be one well-known point in code, called the *main receive* point, where the processes of the family can be scheduled. In our experience this is not always a restriction but rather a good programming practice, anyway. The difference between heavy and lightweight process is transparent to the application code, given that certain design rules are followed, because all the kernel primitives are the same for both types of processes. A process enters the main-receive point by calling the `main_receive` kernel primitive.

Processes communicate by sending messages to each other asynchronously. The primitives supporting message passing are `send`, `receive`, and `main_receive`. From the point of view of a pair of a sender and a receiver process, the DMX kernel guarantees that messages are always received in the same order as they were sent.

1.5.3 Additional Hardware Replication Schemes

In addition to the replication schemes (2*N* and replaceable *N*+1) presented in Section 1.3.1 the DX 200 also supports additional hardware replication schemes which do not use replication of computations and which do not have a unit changeover. Consequently, they will not be in our focus of attention in this thesis and we will present them only for the sake of completeness. The functional units employing these replication schemes are nevertheless controlled by their states, which are: working, blocked, test and separated. The schemes are:

Complementary *N*+1

The computing load created by the application is shared between all available functional units, normally *N*+1 of them. In the case of a unit failure the unit is blocked and taken to the test state. The dimensioned load can as well be taken care of by the remaining units in the case of one or even more of the units been taken out of service. The scheme is called complementary *N*+1 or load sharing.

No replication

There is, in fact, no need to have any redundant units or any replication scheme for functional units of certain types to comply with the availability performance requirements. An example is the exchange terminal for a 2Mbit/s connection, where the probability of failure on the transmission path is expected to be much greater than the probability of failure of the exchange terminal. Consequently, even duplication of exchange terminal hardware would not significantly increase the overall availability performance of the connection. Instead this problem is easily solved by the transmission network and routing arrangements.

1.5.4 Implementation of the System Maintenance

In this section we will present some additional information on the implementation of some of the basic concepts of the DX 200 system architecture, which are necessary to describe the fault-tolerance design of the system.

An additional attribute of the functional unit concept is the *state control mechanisms*. State control mechanisms are mechanisms with which the unit states can be enforced upon the system. They are used by the recovery. These mechanisms are based on hardware and software. From the point of view of the recovery control system, a unit state can be defined in terms of unit state control mechanisms and parameters of these mechanisms. From our point of view, application oriented description of the different states presented in Section 1.3 is more useful and more understandable.

Each functional unit type has its unit state diagram, which defines the possible states and state transitions. To give an example, in Figure 1.9 the state diagram of the 2*N* redundant units (like e.g. the Central Memory unit and the Statistics Unit) is presented. In addition to what already has been said, the transitions between the WO state and testing are shown. They are possible by operator command (from TE to WO) and in the case of an active unit failure (from WO to TE). State transition from WO to SP or to TE always implies a unit changeover.

Of the substates in this thesis we will need the spare-updating (SP-UP) state which is intermediate between a cold stand-by or testing state and the hot stand-by state. This is where the unit is being warmed-up. The hot stand-by state is called spare-executing (SP-EX).

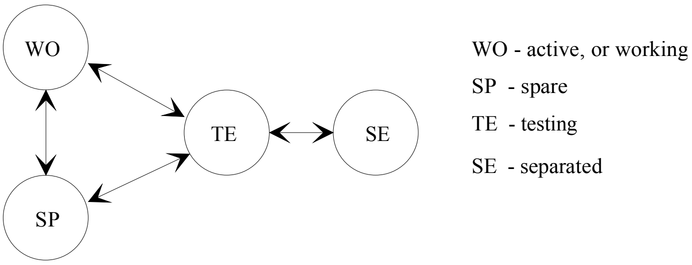


Figure 1.9 State Diagram of the 2*N* redundant units.

As mentioned in Section 1.3 the implementation of the recovery is hierarchical in structure. The central recovery control is much like the conductor and the functional unit level recovery functions are like the members of the orchestra. The central function synchronises e.g. system restart actions and phases ensuring that the order and number of parallel unit restarts is optimal from the overall system point of view. However, if the central function is disabled, the unit level recovery functions are able to attempt unit restart on their own, invoking all the required external services as they are needed. The unit level recovery function controls and supervises all the program families in its unit and orchestrates the recovery actions, like unit restart and unit state transitions at the boundaries of the unit. Because the unit level recovery function controls unit restart and loading, it can also be seen as a functional extension of the operating system. Respectively, the pre-processor level recovery function controls and supervises the processes of the pre-processor and e.g. orchestrates the pre-processor restart and state transition actions.

In addition, applications may have recovery functions, which have to be executed e.g. before or after a unit changeover. Most of these functions are invoked and their sequence is controlled by the unit level recovery.

Not only is the recovery hierarchical by structure, but the recovery actions can also be placed in a hierarchy called *levels of recovery*. The level of recovery is a measure of the amount of disturbance to the system caused by a recovery action when eliminating a failure. There are four levels of recovery which are *no-harm* recovery, *no-premature release* or *minor* recovery, *functional unit failure* or *major* recovery, and *system failure* recovery. No-harm recovery means that the recovery action does not cause any disturbance to calls, no-premature release recovery that existing calls are not disturbed by the recovery action while calls in the set-up phase may be lost. Major recovery means that all calls handled by the functional unit are lost due to the recovery action, and system failure recovery that not less than 50 per cent of all calls in the system are lost due to the recovery action.

2 Definition of the Problem

In this chapter we will first cover the background of the problems of replicated computations in our target system and proceed from there to defining the overall design goals and assumptions to be met by the solution. To justify our choice of the overall approach we will present the relevant internationally accepted telecommunication requirements which we will use to derive the requirements for replicated computations in a switching system. Based on this discussion we will conclude by formulating our overall fault-tolerance approach. The background, the design goals, all the quantitative requirements, and the presentation of many of the qualitative requirements will be DX 200 specific by necessity.

2.1 The problem scope

When the design of the 16-bit DX 200 system started at the end of 1970´s, the spare control computers were put into the hardware design, but at that time the way of their actual usage to preserve the call traffic in the case of a failure of an active unit was not considered. For some of the units the problem was rather simple, but this was not the case with the signalling and call control computers. The issue was first tackled for the signalling units around 1983 and a decision was made to solve the problem in the application code.

The solution was implemented but led to problems which were analysed to be as follows [Ka87]:

- The modularity of the system was violated; the problem was solved similarly but not identically in the different applications.
- Each of the signalling program blocks for subscriber, common channel, and channel associated signalling had a complicated extended automaton structure with states identical to the states of their host computers on the highest level and the signalling protocol states at the lowest. These programs had code for checkpointing the dynamic call state data to the spare computer, repressing charging impulses after the unit changeover, and the most complicated of all, they had code to warm-up the spare. All of this meant that about 30 percent of the signalling program code was there just for the replication of computations. This made the introduction of new software architecture ideas for the call control and signalling applications anything but simple.
- The application development for the 16-bit environment had revealed that the signalling applications were not the only ones that needed replicated computations. Other examples were updating the alarm data to the spare CCS7 signalling unit, some internal protocol applications, semi-permanent connections and the common channel signalling management. The ISDN as well as GSM brought new signalling applications to be developed, also.
- Because of the differences in implementations of replicated computations, the complexity of the system was unnecessarily increasing and thus restricting the ability of the design organisation to generate new end-user applications. It was estimated that the complexity of the software increased as $O(n^2)$, where n was the number of applications that needed replication, instead of linear growth with the number of applications.

- A specific problem with the old implementation was that the signalling applications relied on the statistical unit (STU) as a source of information about the calls to be warmed-up. This tied the STU very closely to the real time and restricted the possibilities of redesigning the statistical and charging data collection programs. This was another example of violation of modularity on the system level.

2.2 Design goals

To solve the problems described in the previous section and to build a comprehensive core system for future application design, the idea of a *virtual machine* was put forward [Ka87, Ka88]. The virtual machine would offer services to all the applications that needed replication of computations, would ensure instantaneous recovery in case of the failure of an active unit, and would support graceful transfer of computations to the spare unit in case of a unit changeover initiated by an operator command. The virtual machine should allow creating, destroying and executing replicated processes. It should also provide for the creation of a hot stand-by spare process for an active one and thus make *graceful unit changeovers* possible. A graceful unit changeover is such that at least all the calls in the ringing or conversation phase are preserved, but calls in the set-up phase may be released after the changeover.

The replicated applications use the services of the virtual machine and the virtual machine itself is controlled by the recovery function of the system according to the fault situation and operator commands. The role of the application designer is to decide whether and how the application has to be replicated and to implement these design decisions using the services of the virtual machine by *high level, declarative language facilities*.

The following general design goals were set:

1. The virtual machine design should support the signalling and call control applications transparently, or at least clearly separate the minimally required replication code from the actual application code. This requirement was of the highest priority because of the amount of continuous design effort which could potentially thus be saved and because of the potential gain in terms of calendar time spend on new signalling and call control applications.
2. The solution should be modular in such a way as to enable changing the hardware replication scheme at least within the existing main categories (2*N* and replaceable *N*+1) transparently to the application code.
3. The solutions should apply to computing systems described in sections 1.2, 1.3 and Figure 1.2. This includes configurations starting from a single pair of computers up to more than a hundred computer units. The solution should apply irrespective of whether the computer system is functionally distributed or all computers execute similar tasks but share the load.

For the DX 200 system the following specific requirements were also put forward:

- It should be possible to start using the virtual machine services for computation replication gradually. At first they would be used only in selected or new applications and only when the overall project deadlines permitted would they be taken into use in the old applications.
- Solutions should also apply to the upgraded old exchanges. Upgrading the old exchanges means that the processor and memory as well as the Message Bus interface plug-in units are changed, but the hardware infrastructure of the exchange remains the same.

- The need to repress charging pulses in the call control applications after unit changeovers should be eliminated by synchronising the real time clocks of all the units and thus making the notion of real time identical in every unit. This would eliminate the problem, because the charging pulses are generated in each of the call control computer units on the basis of the local real time clock.
- The new principles should not rely on information retrieval from the statistical unit and thus should gradually give the design freedom to statistical and charging functions.
- The *level of gracefulness* of unit changeovers remains at least the same as in the application dependent solution. This means that the 2*N* replicated signalling units are capable of graceful unit changeovers in case of the failure of an active unit or if the changeover is initiated by the operator. Signalling units with the replaceable *N*+1 replication scheme have the same kind of graceful changeover, but only if initiated by an operator command.
- *Instantaneous recovery* is an ultimate requirement in the real time environment of switching, if stable calls are to be preserved in a fault situation. This is quite different from what is required in the transaction processing field, where the recovery is enough to be eventual and the systems may thus rely on e.g. roll-back algorithms. What *instantaneous* means in this context will be discussed in Section 2.3.1. The key issue is that the external signal processing delays have to be met for a preserved call after a unit changeover.
- It was required that the warm-up of the spare unit should not take more than *two minutes* measured from the point where all the applications have been started in the spare unit to the point where they are ready for a graceful changeover. This means that an algorithm for copying the dynamic state data of processes from the active unit to the standby unit is required. This algorithm is called the *active warm-up* process. *Passive warm-up*, on the other hand, means that all the new call processing is replicated from the beginning of the warm-up. This situation is also quite different from what one is used to in a transaction processing environment.
- Related to the previous point a performance requirement for the unit changeovers is set. A unit changeover should not take longer than a few minutes. This requirement applies to the unit types that have a unit changeover i.e. the units with 2*N* and replaceable *N*+1 replication schemes.

In all the solution models three assumptions were made. The first is applicable to the class of systems described in Section 1.2. The last two are applicable to all DX 200 switching systems.

1. All the computation replication problems should be solved in software without intricate dedicated and thus expensive hardware support. This *software oriented approach* has been shown to be the most efficient economically [Bo87]. Dedicated hardware support could be used where it would not noticeably increase the cost.
2. It is desirable that the computation replication model does not bring up the need to radically change the basic recovery concepts, such as functional unit or functional unit state. So the model should fit well together with those concepts.
3. It is assumed that the amount of dynamic data to be transferred by the warm-up algorithm may well exceed one Mbyte. Assuming an effective data-rate of 100 kbytes/s from unit to unit over the Message Bus, this would take at least ten seconds to transfer.

2.3 Requirement analysis

In this section we will analyse the specific requirements applicable to the replication of computations in a switching system. Some requirements are based on the characteristics of the process of call processing; others are taken directly from the specifications of the CCITT and from the specifications of the customer organisations.

This analysis is not intended to be very exact. Instead, the goal is to qualitatively justify the critical choices between different possible approaches to solve the problems of replicated computations, to structure the problem scope and thus make it possible to solve the constituent subproblems separately.

In the analysis we will use often use the concepts of reliability and availability. Reliability, *R(t)*, of a system is defined as the probability that the system will not fail within time *t*, given that it was not failed at time 0. Availability, *A(t)*, is defined to be the probability that a system is operational at time *t*. For systems without repair *A(t) = R(t)* [see e.g. Ne87, Chapter 2]. For switching systems a repair time can be assumed and the system may be operational even if failures may have occurred.

2.3.1 Availability and performance requirements

Availability performance, *call processing performance* and *real-time performance* are common key requirements for switching systems. The analysis of these requirements will direct our choice of applicable replication methods and derive a quantitative measure of performance of a replication scheme.

First consider the requirement of *overall availability performance* of the whole system. The mean intrinsic unavailability objective for our specific target system considering complete system failures is three minutes down-time per year [Af]. Experience of systems in operation has shown that this level of availability has been achieved by DX 200 systems with 16-bit control computers [PuAf]. Any new architecture should not compromise these figures.

A definite requirement for the solution is high *system throughput*. On a given hardware the throughput of the class of systems described in Section 1.2 in Busy Hour Call Attempts (BHCA) is determined by the performance of the critical centralised server units and by the performance of the call control and signalling applications executed by the signalling units. In our specific target systems of Figure 1.7 these are the statistics and charging units (STU, CHU), the Marker (M) controlling the switching matrix, and the specific signalling and call control computers.

To come to terms with the real-time performance requirements of switching systems, at least three problems should be considered. First, we will consider what happens, if the call handling processes or more specifically the sending of call related messages through the Message Bus have to be stopped for a short period. A signalling unit sends about 50 messages related to a call to other units. Assuming that the unit is handling 50 000 calls per hour (BHCA), this means that the unit has to send

$$50 \text{ x } 50\,000/3600 \;\cong 700 \text{ messages/s.}$$

As a consequence, if this message traffic is stopped for e.g. 100 ms, the computer units are required to be able to buffer a few hundred messages. This is undesirable, but still possible to implement.

Secondly, there are specific performance requirements for the duration of processing the signalling messages going through the switching system. Such messages may have to be passed through the message bus. Mean values of the shortest durations are required to fall in the range of less than 50 ms or less than 100 ms under the normal reference load (reference load A in [CCITT, Q.543]). Examples of such durations are the *answer sending delay* in the case of in-band line signalling ($\leq$ 50 ms), the *answer sending delay* in other cases ($\leq$ 100 ms), and *exchange signalling transfer delay* ($\leq$ 100 ms).

The third consideration is application transparency. Applications set time-limits for incoming messages. In fact, it is a common fault-tolerance requirement for all applications to be able to recover if an expected message does not arrive in a reasonable time. This is achieved by always setting a time limit for any incoming message. Duration of these time-limits may be as short as 100 ms, while the clock-tick interval is 10 ms. The worst case would be 20-30 ms time-limits.

All these considerations lead us to set an important derived requirement: a single function or application, e.g. the algorithms for computation replication and migration, cannot have exclusive access to the Message Bus for more than *a few tens of milliseconds*. For our problem at hand, it is enough to notice that the expected one Mbyte of dynamic data to be warmed-up clearly cannot be transferred over the Message Bus during this time. Consequently, the system or the computations cannot be stopped for the duration of the warm-up of the spare unit.

An important requirement of a switching system is *very high real-time performance* relative to message passing. The call control, signalling and protocol applications in the switching systems are as a rule very *message intensive*, especially if high capacity and modularity are required at the same time. By message intensiveness we mean the expected proportion of time spent by an application on sending and receiving messages to the total CPU time used by the application. A highly message intensive application may use more than half of its time just sending and receiving messages, at least, if message coding and decoding between the internal variables and the message structure itself is counted as part of message sending and receiving. Note that this measure of message intensiveness is system dependent.

We will use the replication related message count as a measure of performance of the replication scheme. We will judge the applicability of a replication method for a computation on the basis of this measure. For the overall replication scheme we set the requirement that it *should not significantly increase the total message count of the non-replicated computation* to which it is applied. This requirement is intended to ensure that replication of computations does not significantly restrict the maximum achievable throughput of the system on a given level of processor and message bus technology in any system configuration. By significant in this context we mean some tens of percents.

2.3.2 Availability vs. correctness for call processing

Fortunately, the message intensive switching applications are not required to be absolutely reliable. The system may mishandle a single call with a certain probability as long as the whole application behaves reasonably and meets the overall availability performance requirement. In processing a single call, correctness can be sacrificed for the sake of high performance and lower cost of the system as long as certain probability requirements for the availability of service are met. In this sense call processing differs e.g. from transaction processing where it is vital that the database consistency is preserved in all situations and for this reason even heavy performance penalties are acceptable.

The most important of the availability requirements placed on call processing in a switching system is the *premature release requirement*:

R1 The probability of a call being prematurely released in any one minute interval should be less than 2×10^{-5}. [CCITT, Q.543].

Premature release means that an existing call is released before the party assigned to control the release hangs up. This includes calls in both the ringing and the conversation state. This requirement can be seen also as a correctness requirement.

The premature release of a call is avoided, i.e. the level of required correctness may be achieved in a unit changeover, if the *spare computation* is *consistent* with the active one and consequently, can replace the former active computation from the point of view of the environment. We will give the definition of consistency in Chapter 5. The probabilistic requirement for the replicated call processing computation being consistent in the spare computer can be derived from the requirement *R1*, assuming that unit changeover may take place any time and allocating some portion of the whole unreliability to the computation replication scheme. This analysis is done in Chapter 8.

The consistency of a single spare call computation is less important than the instant availability of the service provided by the call processing applications, i.e. applications directly related to setting-up, monitoring and releasing calls. This means that the call is always allowed to proceed even when it is known that the spare computation has failed. It is easy to see that the call processing service availability is better if the active computer is allowed to proceed in case the spare computation fails compared to the case when the whole computation is aborted in the same situation. This is different from e.g. the case of a database, where if the spare computer cannot be updated, a transaction may be aborted to preserve database consistency after a possible changeover. Let us assume that the probability of failure of the spare computation is P_{fs}. If then the whole computation is aborted, the instant service availability is $A_a = 1 - P_{fs}$, provided that other factors are excluded. If on the other hand the active is allowed to proceed, the service availability can be expressed as (again excluding other factors and assuming the failures to be independent):

$$A_p = 1 - P_{fs} * P_{cho} \tag{1}$$

where P_{cho} is the probability that a unit changeover takes place while the spare computation is in the failed state. P_{cho} can be estimated considering the expected duration of the spare computation fail state Δt_{call} and the interval between unit changeovers $\Delta T_{unit\ cho}$.

The worst case when the spare computation may fail is at the call set-up. Then Δt_{call} may be taken as the call holding time, which is usually not more than a few minutes. The interval between unit changeovers $\Delta T_{unit\ cho}$ is expected to be much more than that. Clearly, given that unit changeover may take place any time during $\Delta T_{unit\ cho}$ with equal probability, then P_{cho} is substantially less than 1 and $A_a < A_p$.

2.3.3 Permanence requirements

In some process control systems there are definite *permanence requirements* for data concerning the state of the external world or the process being controlled. Fortunately, in a public telecommunication network all the switching systems have to be able to cope with the crash of a neighbouring switch. Especially in the new signalling systems for digital switching systems provisions are made to recover from such situations by releasing the calls and executing reset or other special protocol procedures. As a consequence, in extreme crash situations we may *rely on the network* being able to recover without exact knowledge about the calls in every network node.

Permanence of the collected statistical and especially charging data as well as the permanence of the database in a switching system have to be preserved. However, it can be argued that in a switching environment the permanence requirement is not as strong as in a transaction processing environment. This is because the users of the switching system database are mostly professionals and the applications using the database are well known, while the transaction processing systems have to be designed for all kinds of users and all kinds of applications.

2.3.4 Charging requirements

The correctness requirement for a charging event is:

R2. The probability of mischarging for a call is less than 10^{-4} per pulse [Finnish PTT].

We use this requirement, because it is tighter than the corresponding requirement of 10^{-4} per call set by the CCITT [CCITT, Q.543]. Clearly, when the charging or statistical data has been collected over a period of time e.g. for a number of calls, the correctness requirement for this data should be much higher than for an event.

The charging pulses for calls are generated by the switching system based on the real-time clock. The requirement for the accuracy of real-time clock synchronisation can be derived from requirement *R2* assuming that

- the system may accomplish one unit changeover in ten minutes,
- unit changeover is an instantaneous action after all the required preparations have been made.

The seemingly very short ten-minute interval between two unit changeovers is assumed because of the desirability of avoiding any user annoyance. On the average the spontaneous unit changeovers occur much more rarely. However, calculating the average on the basis of failure intensity assumptions or data would not be appropriate, because the unit changeovers can also be initiated by operator commands.

Mischarging is caused by the clock skew between the active and the spare computer only immediately after the changeover. The probability that a charging pulse is erroneously sent or received immediately after the unit changeover can be estimated as

$$P_1 = \frac{\Delta c}{\Delta p} \tag{2}$$

where Δc is the clock skew between the two units and Δp is the pulse interval.

As a consequence, the relation for charging pulses between unit changeovers is derived using the requirement *R2*:

$$\frac{P_1}{N_p} < 10^{-4} \tag{3}$$

where N_p is the number of charging pulses in an interval where the unit changeover occurs.

As a consequence, taking into account Eq. 2 and Eq. 3 and the assumed 10 min interval between unit changeovers ($N_p = 10 \text{ x } 60 \times 1/\Delta p$) , we get:

$$\frac{\Delta c}{\Delta p * 60 * 10 * 1/\Delta p} < 10^{-4} \tag{4a}$$

which implies:

$$\Delta c < 60 \text{ ms.} \tag{4b}$$

By looking at the calculation we notice that actually the requirement does not depend on the intensity of call traffic, the pulse interval in a computer or on the interval between clock ticks. The interval between charging pulses in a unit depends on traffic but so does the number of pulses in the unit changeover interval.

If, however, the unit changeover cannot be taken to be instantaneous and during the changeover messages may be misrouted, duplicated or lost, the duration of the changeover (Δx) has to be part of the critical period Δc. The mischarging is actually a problem when it comes to overcharging. When taking this and the possible non-zero duration of the changeover into account, we finally get an asymmetric skew requirement, where asymmetry is a desirable but not an absolutely necessary feature.

$$0 < \textit{SP-clock} - \textit{WO-clock} < 60 \text{ ms} - \Delta x, \text{ where } \Delta x < 60 \text{ ms.} \tag{4c}$$

To reach this level of clock synchronisation accuracy, using dedicated hardware means is naturally the safest way. A software clock synchronisation algorithm, like the one described in [Th89] or used by SIFT [We78], would probably also be appropriate. The clock times have to be sent in broadcast messages to other units, because there is no shared memory. To be on the safe side a combination of hardware and software implementation was chosen for the DX 200, with software for setting the initial time and supervising the clocks and hardware delivering the synchronisation ticks.

2.3.5 MTP and Database requirements

In switching there are a few applications with very high reliability and correctness requirements. An example is the Message Transfer Part (MTP) of the common channel signalling system. For messages carried by the MTP , the following reliability requirements are set [Q.7XX]:

- The probability of a message being lost by the MTP should not typically exceed 10^{-7}.

- The probability of a message being delivered out of the order it was sent should not exceed 10^{-10}.
- The probability of a message data being disrupted and the error going undetected should not exceed 10^{-10}.

These figures place the MTP in a completely different class of applications than signalling and call control. The MTP also has very heavy performance requirements. As a consequence, and because of the existing reliably functioning implementation of the MTP in DX 200, for the time being the MTP was excluded from the problem scope of the envisioned basic virtual machine implementation of the computation replication scheme. However, the MTP could still use some of the replication tools provided by the kernel.

Another application which has clearly more stringent correctness requirements than call control and signalling is the memory resident database system. Also this application could use some of the replication tools of the kernel, but the basic scheme was not required to solve all replication problems of this application.

2.4 Conclusions

The analysis shows that in a switching environment there are a wide variety of applications with largely differing and even conflicting requirements relative to replicated computations. A short summary of the requirements and to what extent they apply to different applications in a switching environment is in Table 1.

Listing other applications would not simplify the picture. The summary suggests employing a structured approach in solving the replication problems.

For the purposes of the discussion on th requirements we will define informally the concepts of *instantaneous correctness*[5] and *eventual convergence*. Instantaneous correctness of the replicated computation is achieved in a state where it can be guaranteed that the state of the spare system is consistent with the state of the corresponding active system. Eventual convergence means that consistency of the spare system is not guaranteed but has a high probability and in the case of the changeover, where the spare system becomes active, the overall system service capability will be recovered although some minor errors may occur in the process. An example of such errors in the changeover is that sometimes a call may be lost.

It does not seem an attractive approach to aim at *instantaneous correctness* at the basic virtual machine level supporting replication of computations because of the following reasons. First, in a switching environment the main application, call control and signalling, does not need absolute correctness in the spare computation. Second, only conditional instantaneous correctness of the spare in both the 2*N* and the replaceable *N*+1 configuration is possible, the condition being that the active does not fail. (Recall that complete single failure masking is possible only if at least three concurrent computations are executed by different computers.) Still another reason is that computation replication schemes like [Co85], [Bl90] aiming at instantaneous correctness seem to have quite a high performance penalty under the conditions prevalent in a switching environment. Two features of the switching environment are important in this sense. The first is the high message intensity of the main applications. Another is that failures are not detected immediately upon occurrence, because of the low level of redundancy acceptable due to economic reasons. We will return to the performance penalty issue in Chapter 10. The same reasoning as to instantaneous correctness applies to the very popular approach of

[5] We will come back to this in Chapter 7.

using atomic broadcast protocols similar to e.g. [Co85], or [Bl90]] on the basic level for reliable message delivery between processes. We will return to this topic also in Chapter 10.

Table 1: The Requirements of Switching Applications.

Application **Requirement**	Call control and signalling	Statistics and charging	Data base	MTP of CCS7
Availability performance of the system	high	high	read: high Else:medium	high
Real-time performance	high	high	read:high; Else: low	high
Correctness of an event	low	medium	medium	high
Correctness of a task	low	high	high	high
Correctness of the application	medium or high	high	high	high
Permanence	No	Yes	Yes	No

The conditions in a system replicated using either the 2*N* or the replaceable *N*+1 principle can be summarised in the form of the following important assumption that has to be taken into account in the design of all the applications and also the replication support.

Assumption 2.1 The *optimistic failure assumption* is that synchronisation failures in the replication scheme and hardware failures are independent of each other and that hardware failures in the active computer are detected by the software fast enough for the recovery to execute the unit changeover before either the failure propagates itself to the other units of the system or the application task is lost because the system cannot react correctly to the events of the external world or maintain its correct state.

Failure is considered to have propagated if the failure of active unit X caused a recovery action of unit Y, e.g. a restart of unit Y or a restart of a program block in unit Y initiated by the unit level recovery function. The condition of preserving the call processing tasks during an active unit failure requires that failure detection times are less than the intervals between call related signalling protocol monitoring events such that will require that both ends of the network signalling link will react in a coherent way to an event. For example, depending on the protocol it may be that if the link is lost, established calls are to be released. Another example is when the called party is ringing and the system is waiting for the answer signal. If a unit involved in the call fails while the call is in the ringing state, the failure has to be detected before the answer signal in order to handle the call correctly. A further reason why a call may be prematurely lost in a unit failure condition is that the unit may not be able to maintain the call related resources in other units of the system as these are to be periodically rereserved to avoid the creation of orphans. Orphans are software or hardware resources such as hand processes and circuits which are reserved and thus can not be reused, although the end-to end service for which these resources were reserved, has been terminated.

The optimistic failure assumption does not provide enough of a basis for complete failure masking, which is the aim e.g. of atomic broadcast protocols. In fact, this assumption requires that *all software applications help with failure detection* in systems that have only limited hardware means for this purpose. It is assumed that the decision that there is a fault is made by the system maintenance, not by any replication tools. The latter are no different from any other applications in that they can only indicate that an error has occurred.

We will call instantaneous correctness under the optimistic failure assumption *conditional instantaneous correctness*.

Eventual convergence and the probability based approaches appear to be a more attractive basis for the solution than building instantaneous correctness to the basic mechanisms of the replication scheme. This approach allows building conditional correctness on top of the basic unreliable replication scheme. This approach aims at meeting high performance requirements while providing the necessary level of correctness required by the signalling and call control applications. In line with this the virtual machine supporting replicated computations need not directly support transaction atomicity, but could still be used as a basis to build the transaction processing subsystem. The computation replication scheme can be viewed as a set of tools. The basic tools should guarantee only some easily achievable level of reliability for replicated computations and the level of reliability could be increased using other tools provided by the kernel. The most advanced tools could even guarantee conditional instantaneous correctness.

Another feature, that would be desirable, because of the real-time performance requirements, is to keep as many as possible of the messages between processes related to call handling *internal to a computer*. If the replication is handled purely on the process level, this would not be achieved since it would lead to a situation, where all the messages would be replicated, and thus *external to the computer*, and would be sent through the message bus in our multicomputer system. This could lead to a considerable performance loss in call processing and would make the message bus interfaces of the signalling units the performance bottle-necks. The problem might be solved by a faster medium and faster interface to that medium but this should be expected to imply some cost penalty. That is why we need a replication concept for entities which are *larger* than processes. This is different from replication schemes known so far [Co85, Bl90].

3 Related Work

A classification of fault-tolerant systems is useful to relate our work to the research that has been done and is going on in other areas. Fault-tolerant systems can be classified e.g. by their application areas, design goals and by specific implementation techniques used in the systems.

Application areas for which fault-tolerant computing systems have been built include *life-critical systems* e.g. for aircraft control, *long-life applications* e.g. for space exploration, *commercial applications*, e.g. transaction processing systems for banking, and *computer and telecommunication network applications*. Somewhat different approaches for fault-tolerance have been adopted in each of these fields, because of the differences in external requirements [see e.g. Ne87].

3.1 Design goals of fault-tolerant systems

For all fault-tolerant systems the objective of *improved availability* is valid. In addition it is useful to characterise the systems by *fault masking capability*, *time for recovery*, *time between maintenance* and *fault coverage*. Systems which mask their faults and have no delays in recovery are said to have high *computational integrity*. In order to achieve very high computational integrity it is necessary to carry out at least two self-checked computations or three non-self-checked computations concurrently [Re84]. However, because of economic reasons in switching, this level of redundancy, as a rule, is not acceptable.

A switching system is not required to mask all its faults. A high probability of correct operation is enough for most of the switching applications and subsystems. In reality, achieving complete fault masking is not possible, because in large real world systems some design faults and – worst of all – some specification faults are always possible. The experience in switching shows that of the reasons for system crashes, namely hardware faults, software errors and human errors, the hardware errors are the least frequent and human errors the most frequent [Mo89]. Consequently, even if all crashes due to hardware failures are eliminated with massive redundancy, the overall availability performance of the systems would not be significantly improved if nothing is done to eliminate design, specification and human errors.

Considering the time to recover, in some cases a switching system is required to recover practically without any delay. This requirement can be expressed e.g. so that stable calls, or calls in the ringing or conversation phase, have to be preserved in some fault situations. High fault masking capability, or *data consistency* which is applicable in our case, is required in handling the semi-permanent data in the exchange. The data consistency requirement applies to the data which is kept in the non-volatile memory. A lower degree of data consistency for the memory resident data is allowed if this leads to no more than occasional failure of some calls and if the situation can be rectified by human action in a simple way using the information and tools provided by the system.

A fault-tolerant system automatically recovers from faults by invoking redundant hardware, but to restore the initial level of reliability, the failed components have to be repaired by human action. Especially in avia-space applications there is often no possibility for human intervention. For these applications *long-life systems* are required. These are usually systems with massive redundancy. Switching systems are, with the exception of satellite based switching systems, all repairable. Performance requirements are set for the reparability. An active repair time of 30 minutes is usually required of the switching systems by the public network operators [Af].

Coverage is a measure of how well the fault-tolerance mechanisms work. It is defined as the conditional probability, given that a fault occurs, that the system will recover properly. A 100% fault coverage is probably impossible to achieve with low-cost designs [Re84], because this would require knowledge of absolute certainty about the faults which will occur in practice. For a switching system e.g. 99% of coverage in fault detection is set. This is not easy to achieve and requires, that all the applications are designed with lots of error handling code. In practice, if e.g. 99.9999% coverage is required, which is the case with life-critical systems, at least three computations have to be carried out concurrently by different computers. In this case knowledge of which faults will occur is not required.

3.2 Fault-tolerance implementation techniques

For *implementing fault-tolerance*, hardware, software, or a combination of both means may be used. Replication of computer units may be used or not. If replication is used, five parameters are involved:

- number of spares related to the number of active units,
- the size of the replication block,
- usage of load sharing,
- how close is the dynamic state of the spare to the active unit, and
- what methods are used to keep the spare unit behaviour consistent with the active one [LA89].

3.2.1 Software vs. Hardware for Fault-Tolerance

The idea of using hardware replication to achieve fault-tolerance and high reliability goes back to von Neumann [Ne56]. Different schemes using the spare computer units have been proposed. Replication of computations can be built into the hardware itself or a software solution can be used. In many other switching systems, e.g. the 5ESS and the Japanese D70 [Cl86, To78, Sn84, Ya89] the hardware approach has been largely used. The hardware approach has the disadvantages, that some dedicated, proprietary hardware is usually required, all the applications get the same fault-tolerance treatment irrespective of their actual needs, and as a consequence, the cost of the control system of the exchange is higher than would be actually needed. Using standard hardware as much as possible has several advantages, some of which are: lesser cost, shorter design cycle, fast application of new hardware technology, less training expenses of the engineering personnel and ease of technology transfer.

There are, however, some fault-tolerance problems which are easy to solve effectively in hardware but very hard in software. Broadcast and multicast protocols on a bus are an example of such problems. Multicasting means sending the same message to a number of recipients. Broadcast means sending to all possible recipients. An *atomic multicast* or broadcast means that the message is delivered to all the non-faulty destinations or to none of them. When implemented in software, the protocols use the hardware ineffectively. The bus interfaces can easily be built to be capable of receiving a broadcast or multicast message from the bus in parallel, so the message has to be sent only once to reach all the destinations practically at the same time.

The software approach to fault tolerance has the advantage that it is more flexible. An example of the flexibility is that only those applications, which really need it, may be executed as replicated computations. Different hardware replication schemes, like 2*N* and replaceable *N*+1 in the DX 200 design, can be incorporated in one software scheme. In a switching system this means that the very inexpensive *N*+1 scheme can be effectively used for call control and signalling on all the trunk interfaces. The software approach may have the disadvantage that replication of computations has to be designed into the applications. So, although the replication scheme may be transparent to the application code, some design-in-the-large effort may be involved to use the scheme.

Consequently, the most cost effective way to solve the fault-tolerance problems seems to be the software oriented approach combined with the use of standard hardware as much as possible. This may include using dedicated hardware support, which may be built using standard components, in carefully selected places. This also means that e.g. the computation replication problem has to be solved in the operating system kernel so that it is as transparent as possible to the applications.

3.2.2 Computation replication schemes

Very little work seems to have been done on studying replicated computations in distributed real-time environments. An exception is *a prototype system* of Natarajan and Tang [NaTa], which contains some interesting ideas. The proposed solution is based on a centralised scheme employing *clones*. The clone is a group of different copies of a process, intended to execute a replicated computation. One of the members of a clone is always the *master* and the others are called *cohorts*. The *choice coordination problem* is the problem of ensuring that all the members of a clone always start executing the same transactions. To solve the choice coordination problem, when the master receives a message, it informs its cohorts about its decision to execute the corresponding transaction with a phase_1 -message. After ensuring that every cohort has received this message, it sends the phase_2 message, and only at this point can the execution start. Consequently, to ensure instantaneous correctness, a high performance cost in the form of executing a two-phase commitment protocol has to be paid. Consequently, when coming to the solution of the choice coordination problem, unfortunately, from our point of view, the solution sacrifices real-time performance in pursuit of instantaneous correctness.

A number of distributed systems have been suggested with software based fault-tolerance features, some of them with replicated computation support and most of them aiming primarily towards transaction processing or avia-space applications: ISIS [Bi85, Bir], Auragen and later MACH [zB90, Bl90, Gl84], Amoeba [Mu86, Re89, Mu90], HOPS [Si89], System V [Ch88], Circus [Co85], Tandem [Ba81], SIFT [We78], Clouds [Da88], Nexus [Tr89], Delta-4 [Sp89], Arjuna [Ar90]. The classical system was the SIFT (Software Implemented Fault-Tolerance) which was designed for aircraft control applications and which used software implemented voting to find agreement upon the results of replicated computations. SIFT produced good research results but was never actually used.

In more recent systems (ISIS, Amoeba, Clouds, HOPS, Nexus, Arjuna) object based programming paradigms have been popular. These systems have primarily been intended to run on workstation hardware connected with a local area network. These systems do not guarantee instantaneous recovery. This is especially the case with schemes like *sender based message logging* [Jo87] and *optimistic recovery* [SY85], in which the idea is to trade recovery time for performance in the absence of failures. This approach is also aimed towards the needs of a computing system based on a network of workstations.

A basic building block of a computation replication scheme is the *message passing scheme*, which describes the configuration of sending and receiving messages between a replicated client and a replicated server. Logically, a very large number of schemes can be suggested. For example, if we take just a paired client and a paired server and look at the possible sending configurations, each of the client members has six possibilities of sending to other members of the configuration excluding sending to itself and sending to own pair only (see Figure 3.1 – arrows indicate messages from the clients to the servers). If the client members are taken to be independent, a total of 36 schemes is acquired. If we take the acknowledgement direction, again 36 schemes are logically possible. So for two-way message exchange, logically 1296 schemes are possible. To design a replication scheme we will have to select the suitable configurations of sending and receiving messages.

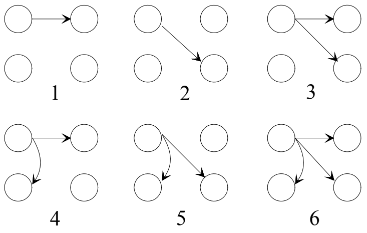


Figure 3.1 Different message passing schemes.

3.2.3 Checkpointing vs. replicated active computations

Two principal architectures of software based replicated computations are *primary/standby* and *modular redundancy* e.g. [Co85, Bi85, Bir]. In a primary/standby scheme, only a single component functions normally; the remaining replicas are on standby in case the primary component fails. The standby processes are passive and the recovery is usually based on the information sent by the primary. Sending the state information of the primary to the standby is called *checkpointing*. Instead of, or in combination with checkpointing, *message logging* may be used. In modular redundancy, each component performs the same function and so all the replicas are active. Voting on the outputs may or may not be used.

To keep the replicas of a computation synchronised Tandem and Delta-4 use a checkpointing scheme [Ba81, Sp89]. For example, passive replicas in Delta-4 are checkpointed transparently in connection with every external message. In Tandem´s Guardian operating system, before each request is processed, the primary sends information about its internal state to the standby in the form of a checkpoint. The checkpoint enables the standby to service the request if the primary fails.

Another example of a primary/standby architecture is the ISIS system [Bi85, Bir]. ISIS implements the concept of replicated objects. In an interaction with a replicated object, one replica plays the role of *coordinator*, and only it performs the operation. The coordinator then uses a two-phase commit protocol to update the other replicas.

In a message intensive switching environment these schemes would clearly be very costly in terms of performance. The problem with checkpointing is that, while the spare components do not make any positive contribution, a very large amount of data has to be transferred to them, especially if they are kept up-to-date all the time. The amount of data can be decreased by allowing the spare to lag behind some of the time. This usually leads to eventual recovery through e.g. a roll-back algorithm, at least if checkpointing is done transparently.

Under the optimistic failure assumption (Assumption 2.1 in Section 2.4), the primary/standby - type components of the replication tools are at best able to tackle the problem of keeping the spare unit consistent with the active one. Clearly then the active computation has to be assumed to be non-faulty and the state of the spare is not independent of the performance of the active unit. This does not, however, mean that the replication tools should give up and not take precautions against hardware failures. It just means that the same external conditions and fault-tolerance design rules apply to the replication support software as to any distributed fault-tolerant software in the system. One of those design rules is that programs should be prepared for hardware failures in their environment and should help with fault detection. Due to the need for a lot of updating from the active computer to the spare, the primary/standby schemes also may *increase the risk of active unit failure propagation* to the spare unit *before the failure is detected*.

The primary/standby scheme is characterised by poor performance in a switching environment and increased risk of failure. The level of achievable consistency is limited due to the low level of redundancy. Consequently, a pure checkpointing scheme as the only means to keep the spare synchronised does not seem very attractive nor economical in real-time switching environments. To ensure instantaneous recovery and, hopefully, to achieve better real-time performance, a scheme with active replicas seems more promising.

3.2.4 The Auragen scheme

MACH is a UNIX compatible operating system kernel that has adopted the Auragen message passing scheme [Bl90]. The idea of the scheme is to save enough information about the computation to be able to eventually recover after a failure of the unit executing the active computation. The scheme is a combination of checkpointing and message logging. The cohort is not active, because it is deemed desirable to save processing capacity of the spare unit. In the scheme every message is sent to three destinations: to the active receiver, to the cohort of the receiver and to the replica of the sender (scheme number 6 in Figure 3.1). The cohort of the receiver buffers the messages, the replica of the sender counts them to track which messages have already been sent and naturally the receiver consumes the message. When there are enough messages in the buffers or enough time has passed, a checkpoint is automatically made. At the checkpoint the state of the active process is copied to its cohort, the queue of buffered incoming messages of the cohort is discarded, and the message count is reset. When a computer fails, the cohort of a lost process is activated. It then runs the lost computation and the outgoing messages are repressed until the message count is reached.

Instantaneous recovery could be added to this scheme e.g. by allowing the cohort to execute normally upon receiving a message and repressing all messages sent by the cohort. The problem with this kind of a scheme is that the replication is still done on the process level. A larger replication entity is needed in a switching environment. Another problem is the performance cost involved with sending the messages to the own replica, too. To minimise the performance cost of replacing simple hardware supported multicast delivery on a bus by two messages over the bus, measures should be taken. A possibility would be that messages on the bus could carry a number of destination addresses recognised by the message bus interface.

3.2.5 Circus

Cooper [Co85] suggested a modular replication scheme for a synchronous message passing environment, in which message passing is hidden by *replicated procedure calls*. A prototype implementation of the scheme was called Circus. Circus is based on the fully symmetric message passing scheme in which all the clients send to all servers and all servers receive from all clients (scheme number 3 in Figure 3.1 for both the active and the spare sender). Voting may be incorporated or not. N-version programming which uses multiple implementations of the same module specification to mask software faults, may be used in conjunction with the scheme. The basic idea on the receiving side is to collect all the incoming messages coming from different members of the sending *troupe* i.e. the set of replicas, and not to send the acknowledgement before all of them have been received, and thus preserve consistent behaviour of the troupe.

In Circus, too, replication is introduced at the process level. The message passing scheme is based on a reliable broadcast protocol and is rather heavy for a message intensive environment. The reliability of message delivery is guaranteed by a compulsory explicit or implicit acknowledgement and the possibility of re-sending the message. The implicit acknowledgement increases performance to some extent, but the existence of the acknowledgement is essential for the replication scheme itself. This is because obviously in a scheme aimed at fault-tolerance, one cannot let the scheme fail or make a decision about a computer unit failure just because of one lost message. The replication scheme is fully transparent to the applications. The message passing scheme is hidden in the replicated procedure call, which is introduced to handle many-to-many communication between troupes. If a failure model without malicious malfunctions is used, voting does not have any positive contribution except in N-version programming. Consequently fully symmetric message passing is used mainly to keep the replicas synchronised. For hiding the replication mechanisms Cooper examines language tools called *stub compilers*. We will use the same idea in Chapter 9.

3.3 Transaction processing systems

In database systems, *transactions* are the means of structuring concurrent computations. Transactions have the essential properties of *atomicity*, *serializability* and *permanence*. Atomicity guarantees that a transaction is an all-or-nothing operation, no intermediate effects of a transaction are ever visible to other transactions. A transaction terminates successfully by *committing*, or unsuccessfully by *aborting*. Before the results of a transaction are committed, they are tentative. If a transaction commits, the tentative effects of the transaction become *permanent* and visible to other transactions. If the transaction aborts, its tentative effects are undone, leaving no trace of ever having been performed. Because the tentative effects of a transaction are not visible to other transactions, if a transaction aborts, there is no *cascading* effect. Achieving atomicity, when more than one machine is involved, requires some form of distributed commit protocol, the best known of which is the *two-phase commit*. Serializability means that the concurrent execution of any number of transactions is equivalent to their serial execution in some order. To achieve this, a *concurrency control algorithm* is required. Most of the algorithms are based on two-phase locking, time-stamps or commit time validation.

An interesting duality between transactions and *conversations* may be established [Ma89]. Conversations are message passing protocols between processes. Consequently there is a duality connection between transactions and message passing schemes.

However, using a transaction model for ensuring the consistent behaviour of the replicas of a replicated computation, has a problem. If a non-consistent behaviour is detected in one of the replicas, what should be done? If the whole computation is aborted, this would mean that higher availability is lost for the sake of instantaneous correctness as we saw in Section 2.3.2. This would not meet our overall goals.

3.4 Conclusion

Replication schemes developed in other application areas and reported in the literature do not seem to satisfy our requirements. Particularly, real-time performance issues and optimisation on the use of hardware have not been considered to the extent which would satisfy the requirements of the switching environment. Often papers on the techniques of replication of computations bypass other aspects of fault-tolerance and other system requirements with minimal attention. It seems typical that correctness is assumed as the predominant requirement without discussion. Due to this unsatisfactory situation we are left with the option of devising our own replication scheme to suit the specific needs of the distributed switching environment.

4 Modelling Tools

To formalise our ideas of replicated computations we shall need a set of models. The *programming model* dealing with the programmer's view of the system was discussed in Section 1.4.1. The *model of computation* defines what it means to execute a program. Additionally a *model of failures* is needed to capture our failure assumptions. On this basis the *model of replicated computations*, i.e. what it means to execute a pair of interchangeable or equivalent computations, will be defined in Chapter 5. Then the *replication scheme*, which is a more or less faithful implementation of the requirements of the model of replicated computations in a system, can be introduced and its properties may be investigated.

4.1 Model of computation

We will first introduce the model informally and then formalise the semantics of the most important concepts using the Asynchronous Communication Tree (ACT) model developed in [Lin91] for SDL systems [Z.100]. We will modify and extend the model introduced in [Lin91] to consider replicated computations in a switching system. We will finish by discussing some properties of computations in a switching system informally.

In a switching system a computation may be *permanent*, that is, the active computation is initiated when the computer unit is taken to the active state and runs until the unit is taken down or fails. From the application point of view such a computation runs forever and the processes for them are allocated permanently and are always ready to proceed, i.e. active. The state of the permanent computations forms a *steady state of the system*. This is a state of the system before any external events have occurred or a state in which the system eventually ends, if all incoming external events are cut out. Because the system collects history information, each new steady state can be expected to be different from the previous one.

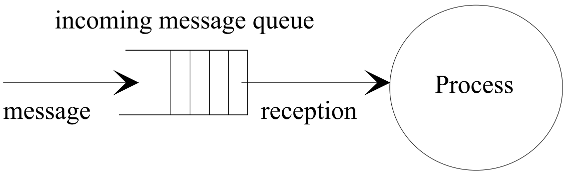


Figure 4.1 Process and its message queue.

A computation may also be of *indefinite length* with *a duration distribution* and an *average*. For such computations, hand processes are allocated at the computation creation time, and released when the computation ends. Indefinite length computations usually *represent the reactions of the system to the events in the external world*. An example of such a computation in a switching environment is the call control.

The semantics of both indefinite length and permanent computations can be described using a language like SDL in terms of *events*, *transitions, processes*, and application process *states* and *actions*. An event is the reception or the sending of a message. *A transition* consists of two elements: a finite sequence of events and the final state. Additionally, actions like *reading* and *writing* a memory resident file and decisions may take place in a transition. *A process* is composed of three elements: initial state, set of receive events, and the transitions. If there is a send-and-receive call inside a transition, this is viewed as a procedure call and is therefore not modelled. In SDL as well as in DMX, arriving messages are put into the *incoming message queue* (see Figure 4.1). When a message is received, it may be processed or saved into the *save queue* of the process. After the reception of the next message the save queue is loaded into the incoming message queue.

The current process state determines, which of the transitions in a process is chosen, when a message is received. The state is an abstraction of the values of variables of a process, which may carry information from transition to transition. These variables are called the *state variables*.

Executing a process means receiving an event, choosing the transition determined by the current state and the received message, executing all the sending events and actions of the transition, and arriving to the final state of the transition. The final state is the new current state. The final state depends on the current state, the received message and the read actions. Variables in the memory resident files which are written by the process are considered an extension of the process state.

We will always assume that a process is *deterministic* in that the final state is uniquely determined by the current state, the received messages, and the values of the data accessed by the read actions. We will also assume that a process is *well behaved* in the sense that it will eventually consume all the messages sent by the user of the service in their order of arrival and will not miss any of them. This also guarantees that the process will send the messages defined by the service which is being produced.

Memory resident files, which are not under the control of the memory resident data base system, are accessed without a buffering system software layer between the files and the applications with the help of a set of library routines mainly to open and close the files and find the data in the files. A process does not necessarily or even usually access all the data in a memory resident file. That is why we will abstract from the implementation and model the data in memory resident files by the concept of *external data units*.

Alternatively, write and read actions could be modelled by send and receive events: this would rid us of the memory resident files and simplify the model. However, there is a difference between read/write actions and an event. An event provides a synchronisation mechanism between the sender and the recipient: knowing the sender, it will be known to us that the sender has finished all the actions that it is supposed to finish before sending the message and nothing that happens after the reception can have an impact on the sender before the message was sent. If a data unit is used for passing information from one process to another, the read and write operations have to be synchronised e.g. by messages or semaphores to make sure that read actions do not occur earlier than the data is made available by the write actions. Another difference is that data in the memory resident files is available until it is overridden by new values i.e. potentially indefinitely, while message passing is typically a fast means of communication. For these reasons we believe that this simplified model would make it difficult to capture the behaviour of our system.

We shall now turn to a more formal presentation. Let us consider a given switching system or any of its subsystems, a *system* for short.

Notation 4.1 The following set symbols are used:

1. Π is the finite set of all Pids, Σ Í Π is the set of Pids of the processes of the system under consideration, and E Í Π is the set of Pids of the processes of the environment. We will typically use s and η to denote processes of Σ and E, respectively.

   We will assume that the system and the environment are disjoint, $\Sigma \cap E = \emptyset$. To consider everything that may happen we will assume $\Sigma \cup E = \Pi$.
2. A is the finite set of *messages*, the alphabet, and A^* is the set of finite strings over A. Messages are typically denoted by a.
3. The mappings <u>sender</u>: $A \rightarrow \Pi$ and <u>dest</u>: $A \rightarrow \Pi$ yield the sender and receiver of some message, respectively.
4. S is the finite set of *process states*. The states are typically denoted by s, suitably subscripted if necessary.
5. D is the set of *data units* of the system and the environment. The data units are typically denoted by d. By D_Σ we will denote the set of data units of the system and by D_E the set of data units of the environment: $D = D_\Sigma \cup D_E$. We will *not* assume that $D_\Sigma \cap D_E = \emptyset$.
6. U is the set of *values* of data units. The set U includes the value *NIL* disjoint from the other values to denote the value of a non-initialised data unit.
7. $V = \{v \mid v: D \rightarrow U\}$ is the set of given mappings from the data units to their values. This set can be extended to the function, also denoted by v, that yields the set of values of any set of data units v: $\mathcal{P}(D) \rightarrow \mathcal{P}(U)$, where by $\mathcal{P}(X)$ we denote the power set of set X. The set of given values of the data units of the system are denoted by $v_\Sigma = v(D_\Sigma) = U_\Sigma$. Mapping v restricted to the data units of system Σ is denoted by $v|_\Sigma$.

Data units are analogous to indefinite length messages, the difference being the method of access to the contents of the data. A process may access several data units during the execution of a transition. All the variables in a memory resident file accessed by a transition of a process are assumed to belong to one data unit from the point of view of that process. Note, also, that the values of data units may be changed by the processes i.e. the mapping from the data units to their values may change.

Notation 4.2 We will need the following mappings from [Lin91, definitions 4.3, 4.4]:

1. <u>input</u>: $S \rightarrow \mathcal{P}(A)$ yields the set of possible input messages of a state.
2. Let the mapping v: $D \rightarrow U$ and $a \in A$ be given.

   <u>output</u>: $S \times S \rightarrow A^*$ yields the string of messages output in a transition initiated by message a from one state to another.
3. The mapping Q: $\Pi \rightarrow A^*$ yields the incoming message queue of a process $\pi \in \Pi$, i.e. $Q(\pi) = q_\pi = <a_1,...,a_n>$ denotes the contents of the queue of process $\pi \in \Pi$ with a_1 being the first element of the queue. By $q_\pi = <>$ we denote that the queue of process π is empty.
4. <u>first</u>: $A^* \rightarrow A$ denotes the first message of a string e.g. of a queue.

We will use mapping output in a context where the values of the data units and the incoming message are given. The alternative definition output: $A \times (D \rightarrow U) \times S \times S \rightarrow A^*$ is cumbersome and would lead to a need to define an initial message for a system which would be an unnecessary burden.

To manipulate queues we need to add elements to the end of the queue and to remove them from the head of the queue. We will adopt the necessary operators from [Lin91].

By (b; i: e) we will denote the mapping that is the same as b except that when applied to the value of i it yields e [Definition 5.1.2 in Gr81]. This allows us to describe new mappings by modifying old ones.

Definition 4.3 Let Q be the queues of processes $\pi \in \Pi$. Let $q_\pi \cdot q'_\pi = <a_1,...,a_n, a'_1,...,a'_m>$ be the concatenation of two queues $q_\pi = <a_1,...,a_n>$, $q_\pi \in Q$ and $q'_\pi = <a'_1,...,a'_m>$, $q'_\pi \in Q$. By $\oplus$ and Θ we will denote the operators for addition and removal of an event to and from a queue:

1. Let $Q(\pi) = q_\pi$, then $Q \oplus \{(\pi, <a>)\} = (Q\ ;\ \pi\text{: } q_\pi \cdot <a>)$
2. Let $Q(\pi) = <a> \cdot q_\pi$, then $Q\ \Theta\ \{(\pi, <a>)\} = (Q\ ;\ \pi\text{: } q_\pi)$.

In our presentation we will ignore the save queues [Z.100, Lin91] and assume that when a message is received it will be consumed before any other event can take place. The save queue semantics has been fully discussed in [Lin91] and would not contribute to the understanding of our problem but would rather only make our presentation more cumbersome.

Definition 4.4 Let the mapping v: $D \rightarrow U$ be given and let $a \in A$ be a message and $s \in S$ a state, then

1. The mapping readset: $S \times A \rightarrow \mathcal{P}(D)$; readset($s$, a) yields the set of data units which may be read in a transition initiated by message a from state s. We will denote the data units in readset(s, a) by $d_{s \times a}$.
2. The function $\underline{\text{read}}_{s\times a} = v\,|_{\underline{\text{readset}}(s,\, a)}$ is defined on readset(s, a) and yields a value for each data unit which a process may read in a transition initiated by message a from state s.

Let the message a be given.

3. The mapping writeset: $S \times S \rightarrow \mathcal{P}(D)$; writeset($s$, s') yields the set of data units which may be written to in the transition initiated by message a from state s to another $s' \in S$. We will denote the data units in writeset(s, s') by $d_{s \times s'}$.
4. The function $\underline{\text{write}}_{s\times s'} = v'\,|_{\underline{\text{writeset}}(s,\, s')}$ where $v' \in V$ is defined on writeset(s, s') and assigns a new value to each data unit which a process may write to in the transition initiated by message a from state s to the new state $s' \in S$.

With the functions readset and $\underline{\text{read}}_{s\times a}$ we can focus our attention on the data units and their values which may be read in one of the possible transitions[6] starting from a state on reception of a message. The mapping writeset and the function $\underline{\text{write}}_{s\times s'}$ are well defined only in the case

[6] See Definition 4.5.

where original values of the data units and the message initiating the transition are given. Alternatively we could have defined a more generic function: *write*: $A \times S \times S \times (D \rightarrow U) \rightarrow (D \rightarrow U)$. We preferred Definition 4.4.4 because working with such a complicated structure is likely to be quite cumbersome and our intention is to use the function only for a given initial state and a given message initiating *the* transition from one state to another. Furthermore, both Definition 4.4.4 and the suggested definition of *write* are not quite enough to capture all the parameters which may have an impact on the new values of the data units. For example, in a real life switching system the new values are dependent on time. The impact of such parameters could be modelled by introducing non-determinism to the model and saying that in Definition 4.4.4 there is a set of possible write functions. We have chosen to ignore such relationships because we are not interested in what the values really are. The closest to the values of the data units we will come is when we ask whether they are equal in the corresponding data units in the active and spare computer units. Our single write function is an adequate representative of the set of write functions for our purposes provided that the same write function is always selected in the active and the spare computer in the transition initiated by a given message from one state to another. We will assume that this is the case.

Definition 4.5 The following mappings are defined (see also [Lin91, definitions 4.2, 4.4]):

1. The state of a system Σ of processes is a mapping $S_\Sigma : \Sigma \rightarrow S$ from the Pids of the processes of Σ to the set of process states.
2. start, initial and current are mappings $\Sigma \rightarrow S$ and denote the starting, initial and current state of a process, respectively.
3. next: $S \times A \rightarrow \mathcal{P}(S)$ is the set of states that can be entered after a message is received in a state and the consequent read operations are executed, i.e. the set of possible follower states.

The next state may depend on the values of the read operations and the mapping next assumes that the alternative values of data units in $\underline{\text{read}}_{s\times a} = v\big|_{s\times a} : D_{s\times a} \rightarrow U_{s\times a}$ can be made available to produce the possible set of next states by executing the system in different environments.

Now we can formally define what we mean by a deterministic process.

Definition 4.6 Let π be a process. Process π is *deterministic* if for a given tuple of (s, a, $\underline{\text{read}}_{s\times a}$), the set $\underline{\text{next}}(s, a) \in \mathcal{P}(S)$ is either a singleton or empty and there is a unique function $\underline{\text{write}}_{s\times s'}$ yielding the values of the data units, the process may have written to while executing the transition initiated by message $a \in A$ from state $s \in S$ to state s'.

Implicit consumption of unexpected messages is a general property inherited by all processes from SDL. This can be presented using the defined mappings by requiring that: $\forall\ \pi \in \Pi$ such that $Q(\pi) = a \cdot q_{\text{tail}}$, s is the current state of process π, and $a \notin \underline{\text{input}}(s)$, event a is consumed implicitly i.e.:

1. $\underline{\text{next}}(s, a) = \{s\}$.
2. $\underline{\text{read}}_{s\times a} = \emptyset$.
3. $\underline{\text{write}}_{s\times s'} = \emptyset$, where $s' = \underline{\text{next}}(s, a)$.
4. New state of the queue: $Q'(\pi) = q_{\text{tail}}$.

An Asynchronous Communication Tree is defined as a tree with the mapping from each of its nodes to a triple that gives the states of the processes of the system, the contents of the incoming message queues of the system and the environment and the contents of the data units of the system. A node of the tree represents the global state of the system. Further, there is the mapping from the arcs of the tree to the observable and silent events of the system. The arcs represent all the possible global state changes of the system. For the sake of simplicity, contrary to [Lin91], we will assume that all the events are observable except those the reception of which was not expected in the current state and will be consumed implicitly.

Definition 4.7 Let $T = (N, \Lambda)$ be a tree with the mapping C which for a node $n \in N$ yields a *configuration* (S_Σ , Q , v) and $\mathrm{l}: \Lambda \rightarrow A \cup \{\tau\}$ where $v \in V|_{D_\Sigma}$ and $\tau \notin A$ stands for a silent event. T is called the Asynchronous Communication Tree, or ACT for short, with the set of nodes N and the set of arcs Λ. For a node of the ACT $n \in N$ the mapping C_S, called the configuration of the processes $\mathrm{s} \in \Sigma$, yields S_Σ, C_Q, called the configuration of the incoming message queues of Σ and E, yields Q, C_V, called the configuration of the data units of the system in some node yields $v|_\Sigma$.

We now have to show how to construct the ACT of a given system and a given environment. The ACT formally defines the semantics of all possible computations which may occur in a system in a given environment. A path of the ACT represents a possible behaviour of the system, i.e. a computation.

Definition 4.8 Let Σ and E be the Pids of the processes of a system and its environment respectively, and let $D_\Sigma \subseteq D$ be the set of data units of the system and let the values of $D_E \subseteq D$ be given. Then $T = (N, \Lambda)$ is an ACT for (Σ, E, D) iff:

1. Initial configuration: The configuration C of the root node $n_0 \in N$ is such that:

   a. $\forall\, \mathrm{s} \in \Sigma$: $C_S\,(n_0)(\mathrm{s}) = \underline{\mathrm{initial}}(\mathrm{s})$,

   b. $\forall\, \pi \in \Pi$: $C_Q\,(n_0)(\pi) = q_\pi = \langle a_1,...,a_n \rangle$ and $\forall\, a_i$, $i = 1,...,n\ \exists\, \pi' \in \Pi$ such that a_i appears in $\underline{\mathrm{output}}(\underline{\mathrm{start}}(\pi'), \underline{\mathrm{initial}}(\pi'))$. Further, if $\langle a_{i_1},...,a_{i_k} \rangle$ are messages in q_π such that $\underline{\mathrm{sender}}(a_{i_1}) = . . . = \underline{\mathrm{sender}}(a_{i_k}) = \pi'$ then the order of their appearance in q_π is the same as their order in $\underline{\mathrm{output}}(\underline{\mathrm{start}}(\pi'), \underline{\mathrm{initial}}(\pi'))$,

   c. $\forall\, \mathrm{s} \in \Sigma$: $C_V(n_0)(\mathrm{s}) = v\,|_\sigma = \underline{\mathrm{write}}_{\underline{\mathrm{start}}(\sigma)\times\underline{\mathrm{initial}}(\sigma)}$.

2. Consumption of a message by the system: Let $n \in N$. $\forall\, \mathrm{s} \in \Sigma$ such that $\underline{\mathrm{first}}(C_Q(n)(\mathrm{s})) = a$, $\underline{\mathrm{sender}}(a) \in \Pi$, $a \in \underline{\mathrm{input}}(\underline{\mathrm{current}}(\mathrm{s}))$[7] and $\forall\, s' \in \underline{\mathrm{next}}(\underline{\mathrm{current}}(\mathrm{s}), a)\ \exists!\, n' \in N$ such that

   a. $C_S\,(n') = (C_S\,(n); \mathrm{s}: s')$,

   b. $C_Q\,(n') = C_Q\,(n) \ominus \{(\mathrm{s}, \langle a \rangle)\} \oplus \{(\underline{\mathrm{dest}}(a_1), \langle a_1 \rangle)\} \oplus . . . \oplus \{(\underline{\mathrm{dest}}(a_n), \langle a_n \rangle)\}$ where $\underline{\mathrm{output}}(\underline{\mathrm{current}}(\mathrm{s}), s') = \langle a_1,...,a_n \rangle$ and the order of messages is preserved as in item (1.b) above,

[7] Note that $\underline{\mathrm{current}}(\mathrm{s}) = C_S\,(n)(\mathrm{s})$, where $\mathrm{s} \in \Sigma$.

c. $C_V(n') = C_V(n) - \{(d, u) \mid d \in \underline{\text{writeset}}(\underline{\text{current}}(\text{s}), s')\} \cup \underline{\text{write}}_{\underline{\text{current}}(\sigma)\times s'}$, where the $\underline{\text{write}}$ assigns the new values to the data units of the $\underline{\text{writeset}}$ of the transition,

d. $\text{l}(n, n') = a$.

3. Consumption of a message by the environment: Let E Í Π be the processes of the environment and let $n \in N$. $\forall\ \eta \in E$ such that $\underline{\text{first}}(C_Q\ (n)(\eta) = a$, $a \in \underline{\text{input}}(s)$, s the current state of process η, and $\forall\ s' \in \underline{\text{next}}(s, a)$ $\exists!\ n' \in N$ such that

a. $C_S\ (n') = C_S\ (n)$,

b. $C_Q\ (n') = C_Q\ (n) \ominus \{(\eta, <a>)\}$,

c. $C_V\ (n') = C_V\ (n) - \{(d, u) \mid d \in \underline{\text{writeset}}(s, s') \cap D_\Sigma\}$[8] $\cup \{(d, u') \mid d \in \underline{\text{writeset}}(s, s') \cap D_\Sigma\}$, where u' are the new values written in the transition by the process of the environment to the data units of the system,

d. $\text{l}(n, n') = a$.

4. Implicit consumption of a message by the system or the environment: Let $n \in N$. $\forall\ \pi \in \Pi$ where $\Pi = \Sigma \cup E$, $\underline{\text{first}}(C_Q(n)(\pi)) = a$, s the current state of process π, $a \notin \underline{\text{input}}(s)$ $\exists!\ n' \in N$ such that

a. $C_S\ (n') = C_S\ (n)$,

b. $C_Q\ (n') = C_Q\ (n) \ominus \{(\pi, <a>)\}$,

c. $C_V(n') = C_V(n)$,

d. $\text{l}(n, n') = \tau$.

The first item of Definition 4.8 tells how to construct the initial configuration in which each process is in its initial state after having executed the transition from the start to its initial state. Messages sent by a process in this transition are put into the incoming message queues of the destination processes in the order they were sent. Messages sent by different processes to a process are queued in a random order. The initial contents of the set of data units is produced by write operations in the transitions from start to the initial state. For simplicity, here we ignore the order of write operations. We also ignore the fact that some of the data in the system are loaded from non-volatile memory or originate from before the system restart.

The second item of Definition 4.8 deals with consumption of a message by a process of the system. The message may be sent by a process of the environment or by a process of the system. Internal events in the system affect the future behaviour of the system with respect to the environment because the resulting state changes have an effect on what messages are expected from the environment. It is implicit that the next state of the system may depend on the values of the data units which are read by the process and which may be data units of the system or the environment (see Definitions 4.4 and 4.5). The mapping $\underline{\text{next}}$ was defined in such a way (Definition 4.5.3) that the same set of new nodes $n' \in N$ of the ACT result in any given data environment. It is assumed that any time a message is expected, it also eventually arrives or alternatively a time-out is active and will elapse, leading to a retry, other recovery or release procedure. It is also assumed that all the values of the data units which are read by the process and may affect the next state of the process, will be available by executing the system in

[8] Note that D_Σ is the domain of mapping $C_V(n)$.

different environments. As a consequence, all possible ways the system may evolve in a given environment are covered by the ACT. The order of messages sent by the transition is preserved in the same way as in item (1) of the definition. The data units which are produced (written) by the transition become members of the set of data units of the system. For simplicity, the model ignores the combined order in which the write actions and sending of messages are mixed in a transition. This is justified by the assumption that processes are deterministic and consequently, if data is passed from one process to another through a data unit, this information exchange is synchronised by a message.

The third item of Definition 4.8 deals with the consumption of messages by the environment. This takes care of the deletion of messages from the queues of the processes of the environment. In sub-item (3.c) *a data unit of the system may be written to by a process of the environment*. As the values of the data unit will be changed and the data units are part of the global state of the system, the mapping from the data units of the system to their values is modified without any of the processes of the system being directly responsible.

The fourth item of the definition takes care of the implicit consumption of unexpected messages by the processes of the system as well as by the environment.

A process of the environment may read a data unit of the system or consume a message sent by the system. These events do not, however, have any impact on the ACT of the system other than described in items three and four of the definition, but may affect the state of the environment. The possible later impact on the messages sent by the environment to the system or to data units read by the system is taken care of by item two of the definition. Write operation by the processes of the environment to D_E such that $D_{\Sigma} \cap D_E =$ Æ are not described by the model because they do not have a direct impact on the state of the system. If a process of the system reads data units from such D_E this is taken care of by item two of the definition. The definition describes the semantics of executing a transition as if the transition is executed until the final state before the messages sent in the transition are processed by the destination processes. This cannot have any impact on the execution of the rest of the transition provided that after sending a message the process does not read any data units of the environment and the destination process does not read any data units which the sender may write in the rest of the transition. Due to these considerations we will assume that this simplification is justified. In fact the execution order of transitions linked by a message may not be well defined, because scheduling may be possible during a transition. For example, this is the case in the DX 200 system.

As such the model described in Definition 4.8 does not cover the creation and deletion of processes. We will come back to this issue in Chapter 5.

For the purpose of modelling replicated computations, the concepts of *isomorphic global states* and *isomorphic behaviour* of processes and sets of processes are useful.

Isomorphic behaviour implies that there is a bijection for renaming the processes and a bijection for renaming the data units such that the configurations and the whole ACTs of the isomorphic systems (Σ_1, E_1, D_1) and (Σ_2, E_2, D_2) are the same except for the renaming.

Definition 4.9 Let Σ_1 and Σ_2 be systems of processes, $T_1 = (N_1, \Lambda_1)$and $T_2 = (N_2, \Lambda_2)$ the ACTs for (Σ_1, E_1, D_1) and (Σ_2, E_2, D_2) and C^1 and C^2 their configurations, respectively.

1. A global state $n_1 \in N_1$ is isomorphic to a global state $n_2 \in N_2$ iff $\exists$ bijection r: $\Sigma_1 \to \Sigma_2$ such that $\forall$ $s_1 \in \Sigma_1$, $s_2 \in \Sigma_2$ $r(s_1) = s_2$ and r: $D_1 \to D_2$ such that $\forall$ $d_1 \in D_1$, $d_2 \in D_2$ $r(d_1) = d_2$ and

   a. $r(C^1{}_S (n_1)) = C^2{}_S (n_2)$, and

   b. $r(C^1{}_Q (n_1)) = C^2{}_Q (n_2)$, and

   c. $r(C^1{}_V (n_1)) = C^2{}_V (n_2)$.

   We will denote such global states $C^1 (n_1) \simeq C^2 (n_2)$, systems of processes $\Sigma_1 \simeq \Sigma_2$ and sets of data units $D_1 \simeq D_2$.

2. The behaviour of the system of processes Σ_1 and its data units D_{Σ_1} is isomorphic to the behaviour of the system of a processes Σ_2, and its data units D_{Σ_2} iff,

   a. the initial configurations of T_1 and T_2 are isomorphic: $C_1 (n^0{}_1) \simeq C_2 (n^0{}_2)$, and

   b. $\forall$ $a \in A, n_1, n'_1 \in N_1, n_2 \in N_2$ $\exists$ n'_2 such that

   $(C_1 (n_1) \simeq C_2 (n_2)) \Rightarrow ((C_1 (n'_1) \simeq C_2 (n'_2))$

   where n'_1 is any next node of T_1 according to Definition 4.8, and

   $\forall$ $a \in A, n_1 \in N_1, n_2, n'_2 \in N_2$ $\exists$ n'_1 such that

   $(C_1 (n_1) \simeq C_2 (n_2)) \Rightarrow ((C_1 (n'_1) \simeq C_2 (n'_2))$

   where n'_2 is any next node of T_2 according to Definition 4.8, and

   c. $l_1(n_1, n'_1) = l_2(n_2, n'_2) = a$ or $l_1(n_1, n_1') = l_2(n_2, n_2') = \tau$.

   We will denote such isomorphism by $T_1 \simeq T_2$.

If the behaviours of two processes are isomorphic until event $a \in A$, it implies that they are in the same state when the event occurs. This is true assuming that we do not have the means to differentiate between two states of a process unless a message was received or sent or an action executed by the process. The opposite, however, is not true. Two processes can naturally be in the same state even if their histories are different. Definition 4.9 gives a criterion to an external observer to decide whether two processes are *observably* different or not by comparing sets without the identity of the processes playing a role.

Two systems of processes with isomorphic behaviour may be distinguished at their external interface by differing Pids in the external messages. Two otherwise identical processes may have different Pids because it is the responsibility of the service provider to allocate the process which produces the service. Two *computations* are said to be *disjoint* if they are executed by disjoint execution groups i.e. by groups without common members. Disjoint computations may be isomorphic.

The behaviour of a deterministic process is isomorphic to the behaviour of a *process of the same type* if the initial states are identical, the processes are *exposed to the same sequence of* *receive events* and the *values of the isomorphic data units of the environment are identical* at the moment they execute the read actions. Using our model of computation we can formalise these notions.

Definition 4.10 Let us denote the mappings of a process by superscripting the name of the mapping by the process Pid, e.g. $\underline{\text{input}}^{\text{s}}$ is the input mapping of process s. Process s_1 is of the same type as another process s_2 iff in any given environment:

1. the processes have the same set of states: $S^{\text{s}_1} = \{s_i^{\text{s}_1} | \ i = 1,...,n\} = S^{\text{s}_2} = \{s_i^{\text{s}_2} \ | \ i = 1,...,n\} = S$,
2. the processes have the same mappings of input messages: $\underline{\text{input}}^{\text{s}_1} = \underline{\text{input}}^{\text{s}_2}$, and
3. the processes have the same mappings for the next state: $\underline{\text{next}}^{\text{s}_1} = \underline{\text{next}}^{\text{s}_2}$, and
4. they send the same messages in the corresponding transitions, i.e. the output mappings are equal: $\underline{\text{output}}^{\text{s}_1} = \underline{\text{output}}^{\text{s}_2}$, and
5. the sets of data units read by the processes are isomorphic:
   $\forall \ s \in S$ and $\forall \ a \in \underline{\text{input}}^{\text{s}_1}(s)$: $\underline{\text{readset}}^{\text{s}_1}(s, a) \simeq \underline{\text{readset}}^{\text{s}_2}(s, a)$, and
6. the sets of data units the processes write into in the corresponding transitions are isomorphic, i.e. $\forall \ (s, s') \in S \times S$: $\underline{\text{writeset}}^{\text{s}_1}(s, s') \simeq \underline{\text{writeset}}^{\text{s}_2}(s, s')$,
7. for the data units it applies that $\forall \ s \in S$, $\forall \ a \in \underline{\text{input}}^{\text{s}_1}(s)$ and $\forall \ s' \in \underline{\text{next}}^{\text{s}_1}(s, a)$:

   $(\underline{\text{read}}^{\text{s}_1}{}_{,s \times a}(d^{\text{s}_1}) = \underline{\text{read}}^{\text{s}_2}{}_{,s \times a}(d^{\text{s}_2})) \Rightarrow (\underline{\text{write}}^{\text{s}_1}{}_{,s \times s'}(\delta^{\text{s}_1}) = \underline{\text{write}}^{\text{s}_2}{}_{,s \times s'}(\delta^{\text{s}_2})$ where $d^{\text{s}_1} \in \underline{\text{readset}}^{\text{s}_1}(s, a)$ and $d^{\text{s}_2} \in \underline{\text{readset}}^{\text{s}_2}(s, a)$ are isomorphic and $\delta^{\text{s}_1} \in \underline{\text{writeset}}^{\text{s}_1}(s, s')$ and $\delta^{\text{s}_2} \in \underline{\text{writeset}}^{\text{s}_2}(s, s')$ are isomorphic.

Lemma 4.11 Let s_1 and s_2 be two deterministic processes of the same type and let initial configurations of the ACTs $T_1 = (N_1, \Lambda_1)$ and $T_2 = (N_2, \Lambda_2)$ for (s_1, E_1, D_1) and (s_2, E_2, D_2) be $C(n^0{}_1)$ and $C(n^0{}_2)$ respectively. Let $s \in S$ be a state of the processes s_1 and s_2, let $a \in \underline{\text{input}}^{\text{s}_1}(s)$. If

1. $C(n^0{}_1) \simeq C(n^0{}_2)$, and
2. the processes are exposed to the same messages,
3. there are no processes in the environment such that they write into the data units of the systems (s_1, E_1, D_1) and (s_2, E_2, D_2), and
4. $\forall \ s \in S$ and $\forall \ a \in A$: $\underline{\text{read}}^{\text{s}_1}{}_{,s \times a}(d^{\text{s}_1}) = \underline{\text{read}}^{\text{s}_2}{}_{,s \times a}(d^{\text{s}_2})$ where $d^{\text{s}_1} \in \underline{\text{readset}}^{\text{s}_1}(s, a)$ and $d^{\text{s}_2} \in \underline{\text{readset}}^{\text{s}_2}(s, a)$ are isomorphic,

then the behaviours of processes s_1 and s_2 are isomorphic.

Proof: We must show that under the conditions of the lemma:

a. $\forall \ a \in A$, $n_1 \in N_1$ and $n_2 \in N_2$: $(C_1(n_1) \simeq C_2(n_2)) \Rightarrow (C_1(n_1') \simeq C_2(n_2'))$ where n_1' and n_2' are the next nodes of the ACTs according to Definition 4.8, and:

b. $\text{l}_1(n_1, n_1') = \text{l}_2(n_2, n_2') = a$ or $\text{l}_1(n_1, n_1') = \text{l}_2(n_2, n_2') = \tau$.

Let us assume that $C_1(n_1) \simeq C_2(n_2)$ and let us denote the current state $C_S(n_1)(\mathrm{s}_1) = C_S(n_2)(\mathrm{s}_2)$ by s. According to Definition 4.10, item (2): $\underline{\mathrm{input}}^{\mathrm{S}_1}(s) = \underline{\mathrm{input}}^{\mathrm{S}_2}(s)$. Consequently, $a \in \underline{\mathrm{input}}^{\mathrm{S}_1}(s) \Rightarrow a \in \underline{\mathrm{input}}^{\mathrm{S}_2}(s)$.

According to Definition 4.8 $\forall\, a$:

I. if $a \in \underline{\mathrm{input}}^{\mathrm{S}_1}(s)$ and $a \in \underline{\mathrm{input}}^{\mathrm{S}_2}(s)$ then

i. There is a single new state due to the fact that the processes are deterministic and we can denote this new state by $s' = C_S(n_1') = \underline{\mathrm{next}}(s, a)$.
$C_S(n_1') = C_S(n_2')$ due to Definition 4.10, items (3) and (5) and conditions of the lemma. Alternatively $\underline{\mathrm{next}}(s, a) = \varnothing$ and there is nothing to prove.

ii. $C_{Q\mathrm{s}_1}(n_1') = C_{Q\mathrm{s}_1}(n_1)$ O,- $\{(\mathrm{s}_1, <a>)\} \oplus \{(\underline{\mathrm{dest}}(a_1), <a_1>)\} \oplus \ldots \oplus \{(\underline{\mathrm{dest}}(a_n), <a_n>)\}$ where $\underline{\mathrm{output}}^{\mathrm{S}_1}(s, s') = <a_1,\ldots,a_n>$; due to our initial assumption $C_{Q\mathrm{s}_1}(n_1) \simeq C_{Q\mathrm{s}_2}(n_2)$ and due to item(4) of Definition 4.10 and the fact that $\{(\mathrm{s}_2, <a>)\}$ is also removed from $C_{Q\mathrm{s}_2}(n_2)$: $C_{Q\mathrm{s}_1}(n_1') = C_{Q\mathrm{s}_2}(n_2')$.

iii. $C_{V\mathrm{s}_1}(n_1') = C_{V\mathrm{s}_1}(n_1) - \{(d, u) \mid d \in \underline{\mathrm{writeset}}^{\mathrm{S}_1}(s, s')\} \cup \underline{\mathrm{write}}^{\mathrm{S}_1},_{s\times s'}$. Due to our initial assumption $C_{V\mathrm{s}_1}(n_1) \simeq C_{V\mathrm{s}_2}(n_2)$, and due to Definition 4.10, items (5), (6) and (7):

(5): the sets of data units read by the processes are isomorphic: $\underline{\mathrm{readset}}^{\mathrm{S}_1}(s, a) \simeq \underline{\mathrm{readset}}^{\mathrm{S}_2}(s, a)$; further due to condition (4) of the lemma the values read in the isomorphic data units are equal,

(6): the sets of data units that may be written to by the processes are isomorphic: $\underline{\mathrm{writeset}}^{\mathrm{S}_1}(s, s') \simeq \underline{\mathrm{writeset}}^{\mathrm{S}_2}(s, s')$, and

(7): the values that may be written into these isomorphic data units are equal and consequently, $C_{V\mathrm{s}_1}(n_1') \simeq C_{V\mathrm{s}_2}(n_2')$.

iv. $\mathrm{l}(n_1, n_1') = a$ and due to conditions of the lemma $\mathrm{l}(n_2, n_2') = a$.

II. if $a \notin \underline{\mathrm{input}}(s)$ and $a \notin \underline{\mathrm{input}}(s)$ then

i. $C_S(n_1') = C_S(n_1) \simeq C_S(n_2') = C_S(n_2)$.

ii. $C_{Q\mathrm{s}_1}(n_1') = C_{Q\mathrm{s}_1}(n_1) \ominus \{(\mathrm{s}_1, <a>)\}$ and the same a will be removed from the queue of process s_2 .

iii. $C_{V\mathrm{s}_1}(n_1') = C_{V\mathrm{s}_1}(n_1) \simeq C_{V\mathrm{s}_2}(n_2') = C_{V\mathrm{s}_2}(n_2)$.

iv. $\mathrm{l}(n_1, n_1') = \mathrm{l}(n_2, n_2') = \tau$.

Consequently, due to I.(i, ii, iii) and II.(i, ii, iii) the above requirement (a) is satisfied and due to I.(iv) and II.(iv) requirement (b) is satisfied. According to conditions of the lemma, item 3 of Definition 4.8 may not change the state of the systems. Consequently, the lemma follows.

In modelling graceful software updating procedures, subsets of the set of receive events of a process are of interest. Based on this, upward compatibility issues can be analysed. For our purposes, however, the definition given is sufficient.

The system functional requirements specification, which is a step in the system design process, starts with the identification of the external functions or services of the system, like the operator commands and services offered by the switching system into the network. This is followed by the implementation specification phase starting with the breakdown of the functionality into the necessary software and hardware components of the system which together will implement the external functions. Finally comes the specification of the behaviour of the components.

To support these ideas and to preserve the overall system view it is useful to define concepts which allow us to talk about the behaviour of larger portions of the system than just a process. For this purpose we will introduce the concept of an *execution group*. We will apply this concept in Chapters 5, 6 and 7.

Definition 4.12 Let Σ be a system of processes s, s' $\in \Sigma$ and let $T = (N, \Lambda)$ be an ACT for a subsystem of processes $\Gamma \subseteq \Sigma$. Any such set Γ is an *execution group* iff

1. Γ has only one member s $\in \Gamma$, or

2. $\Gamma = \Gamma' \cup \{s\}$, where s $\notin \Gamma'$ such that Γ' is an execution group and
   $\exists$ s' $\in \Gamma'$ such that $\exists\, n, n' \in N$ and $a \in A$: $l(n, n') = a$ and
   i. sender(a) = s and dest(a) = s' , or
   ii. sender(a) = s' and dest(a) = s.

The definition implies that the set is always connected and may not be a fragmented set. The global state of an execution group includes the values of the data units which may be written to by any of the processes of the group. A process may be a member of many execution groups. The concept of an execution group is a tool that allows us to focus our attention on specific sets of processes for which a common condition applies and which are of interest to us.

The largest possible execution group in a system contains all the processes of the system. However, by restricting ourselves to a smaller group of processes we may talk about e.g. the properties of a system that handles just one call. For this purpose an interesting execution group might include all the processes participating in handling a call or all processes which store or access call state variables. We may also want to restrict such a group further by excluding e.g. processes which are solely devoted to charging and statistics. Another interesting execution group might be the set of processes which carry out an operator command.

From the point of view of an execution group, messages may be *external* or *internal*. Messages passed between the members of the group are internal. Messages sent or received by a process in the group to or from any process which is not a member of the group are called external messages.

Informally all that is initiated and happens in an execution group upon arrival of an external message is called a computation. So, informally we can see a computation as a "service" produced by an execution group.

A finite computation is represented by a finite path of the corresponding ACT. Of interest are finite computations which are executed by an execution group *as a reaction to a single external event*.

Definition 4.13 A finite computation resulting from a single external event in an execution group is called a *reaction*. An execution group is *reactive*, if isolated from the reception of external messages from a state onwards, it will after a finite computation end up in a steady state where it will not change state unless a new external message is received.

By definition the first arc of the ACT of a reaction is labelled by the external event. All the rest of the arcs of the ACT are labelled by internal messages of the execution group. We will use these concepts in Chapters 5 and 7.

The transitions of a *reaction* may send messages to the environment. A reaction ends in a *steady state* represented by a node of the ACT with empty incoming message queues if the execution group is isolated from the external incoming messages after the initial event was received and if there are no processes in the environment which write into the data units of the execution group. If there is no isolation, according the service definition, the reaction may be interrupted by a new external event before the execution of the reaction ends. When a reaction ends in the steady state, i.e. all internal messages are consumed, we say that the *reaction is complete*.

A system is said to be *closed* if there are no processes in the environment which write into the data units of the system.

In the sequel we will discuss some additional concepts informally. These concepts will not be used in further formal definitions but will help to describe the behaviour of a switching system intuitively.

A special case is a *sequential reaction*, which may be visualised as a chain of transitions. This means that during the computation, each process in the execution group invokes not more than one service provider at a time and if a process executes more than one transition in the *reaction*, they are different transitions. In terms of the structure of the ACT this means that the *degree of the nodes* of the ACT of the system is not more than three, i.e. a node may be an endpoint to the arc for entering into the node, to the main outgoing arc for leaving the node and possibly to the additional arc labelled with the time-out message to avoid deadlock. Note that this does not require that the transition of the user is finished before the provider starts execution.

A *computation is sequential* if it is a sequence of *complete sequential reactions*. A sequential reaction *either ends when it is complete or it is interrupted* by a new external message, because the restriction of invoking no more than one service at a time applies to external and internal services alike.

An example when the concept of an *interrupt* (a new external message arrives before a reaction is complete) is useful, is to describe the behaviour of the call control system that has to accept a disconnect message from the initiator of the call while the system is waiting for an answer signal from the called party. To describe this situation at least two main outgoing arcs of the ACT are needed in the node representing such a global state.

Involvement of a process in a computation may be *stateless* or *state oriented*. If the state variables of a process have at least two different values, because of the process being involved in a computation, the involvement is state oriented. In terms of the ACT this means that the configuration of the state variables C_S or the configuration of the data units C_V is not constant during the path representing the computation. This includes all cases in which the process writes anything to its state variables or to memory resident files. The subset of the processes of an execution group, the involvement of which in a computation is state oriented, is called the *state oriented execution group*. When the values of the state variables of a process do not change because of the computation, the involvement of the process is stateless. An example of a stateless involvement is a process offering database read services to call control.

Control in the computation can flow in different directions in an execution group. Each *reaction* executed by a state oriented execution group creates an order of involvement of the member processes. This order we will call the *reaction order*. It is not necessarily the same as the execution order of the corresponding transitions, which depends on *priorities*. Priorities are process attributes that are used to determine the execution order of processes which are ready for execution. In terms of the ACT an order of involvement is produced by applying the function <u>dest</u> to all the labels on the arcs of the path representing the computation.

An example is the order in which the processes were initially allocated for the computation. In this order there is only one process, which is the *topmost user* or initiator of the computation. If we look only at the uses-provides relations between the processes we see that the *allocation order* is a partial order on processes. Even more precisely, it is a tree of processes, because (1) a user may have several service providers in this order, and (2) each service provider may be allocated only once, so cycles are not possible. There may be several *downmost service providers* in this example computation and some processes may have both the role of a user and the role of a provider.

4.2 Model of Failures

A hierarchy of models of faults and failures is required to build a system which is highly available, able to detect its own faults, recover from them, and locate the faults. For purposes of fault location a low level model dealing with the behaviour of a faulty circuit or a faulty chip is needed. This model is based on the physical fault model. For the purposes of recovery and replicated computation modelling, however, a higher level failure model is more useful. This model deals with the behaviour of a faulty computer unit.

A computer unit failure may be either *a crash* during which a machine simply ceases to function, or *a malfunction* during which it functions incorrectly. A malfunction may be *malicious* or *non-malicious*. If the model incorporates malicious malfunctions, by which is meant a behaviour which is seen differently by different components of the system, a solution to the *Byzantine agreement* problem must be implemented. The Byzantine agreement problem was first identified and solved by the researchers who tried to prove the correctness of the SIFT system [Pe80, We78]. The problem is as follows. A sending processor wishes to communicate some value to each of the *n* receiving processors. The sender may malfunction, sending different values to different recipients, or not sending anything to some of them. However, the following conditions must hold:

1. All correctly functioning recipients agree on the same value.
2. If the sender is functioning correctly, then all correctly functioning recipients agree on the value sent.

By a non-malicious malfunction we mean the class of failures where the possibility of sending different values to different recipients in the Byzantine assumption is excluded, but still messages may be lost. These failures can be handled without the full Byzantine agreement problem solution. An example is when a multicast message is correctly received by some of the recipients but not received at all by at least one of them. When non-malicious malfunctions are incorporated, the classes of failures must be specified more exactly each time.

The Byzantine failure assumption is not very close to reality and may easily lead to inefficient systems, because of the complicated agreement protocol. Known alternatives to the Byzantine assumption are the notion of a *fail-stop processor* [Sc83, Sc84] and the notion of a *fail-silent processor*[SSh90]. The idea in these concepts is that a processor may crash, but will never malfunction. If it is possible to detect malfunctions, then an ordinary processor can be transformed into a well behaved fail-silent processor by causing it to halt whenever a malfunction occurs. Unfortunately, fault detection is a very hard problem, too.

In this thesis a compromise is chosen. The basis is the fail-silent processor assumption. It is assumed that if a message is received, its validity can always be established. Messages may be lost, but this can be detected e.g. by time-outs. So, the possibility of some non-malicious malfunctions is also incorporated into the model.

It should be noted that in our case the "fail-silence" abstraction is created from ordinary processors by extensive fault detection and recovery software instead of duplicating the processors to form fail-silent nodes as in [SSh90]. As we defined in our *optimistic failure assumption* (Assumption 2.1 in Section 2.4), the failure detection is assumed to take some time, but to be fast enough to stop failure propagation into other processors of the system. It is our conviction that this model is close enough to reality in a distributed switching environment with a low level of redundancy and it can therefore help to come up with a more efficient and practical implementation of the replication of computations.

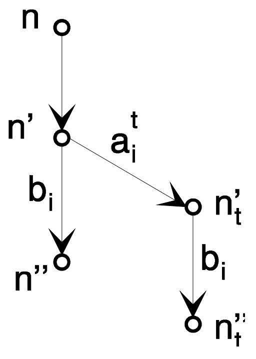


Figure 4.2. An extract of an ACT

The requirement of the optimistic failure assumption can be presented in terms of the protocol for message exchange between processors using our model of computations (see Figure 4.2). In the presentation we will assume that the processes of the environment run on a different processor than the system and that that particular processor is duplicated and may fail. The presentation is not a formal model of failures; it only expresses what is the impact of the optimistic failure assumption to the ACT of a system that can recover from a failure by a unit changeover.

Let Σ be a system, E the environment, $\eta \in E$ a process of the environment, A the alphabet of messages and $T = (N, \Lambda)$ the ACT for (Σ, E, D). Let us look at an extract of system behaviour in which a process of the system $\sigma \in \Sigma$ sends a message a_i to a process of the environment and expects an acknowledgement b_i. Let n, $n' \in N$, and let $\exists\, a \in \underline{\text{input}}(C_S(n)(\sigma))$ such that $\lambda(n, n') = a$, then $\forall\, a_i \in <a_1, \ldots, a_m> = \underline{\text{output}}(C_S(n)(\sigma), C_S(n')(\sigma))$ and $\underline{\text{dest}}(a_i) = \eta \;\; \exists\, a_{i'}^{t}$ such that

1. $a_{i'}^{t} \in \underline{\text{input}}(C_S(n')(\sigma))$,

2. $a_{i'}^{t}$ will arrive not later than $\Delta t_{a_i} + 1$ and not earlier than Δt_{a_i} where one represents the clock tick interval,

3. $\exists\, n'_t \in N$ such that $\lambda(n', n'_t) = a_{i'}^{t}$ and $a_i \in <a'_1, \ldots, a'_k> = \underline{\text{output}}(C_S(n')(\sigma), C_S(n'_t)(\sigma))$ and $\underline{\text{dest}}(a_i) = \eta$,

4. $\exists\, n''$, $n''_t \in N$, such that $\lambda(n', n'') = \lambda(n'_t, n''_t) = b_i$ where b_i is the acknowledgement message for a_i: $\underline{\text{sender}}(b_i) = \eta$.

To fulfil requirements of the optimistic failure assumption we will assume that the time for failure detection and unit changeover Δt_F is:

$$\Delta t_F < \Delta t_{a_\mathrm{i}}.$$

Faults may be permanent, transient, or intermittent by their *time behaviour*. A *transient fault* causes an error to occur but disappears before it can be located. This is modelled by assuming the transient fault to have a duration with some distribution [Yi80]. Intermittent faults are faults with some occurrence period. In a switching system, it is assumed that transient and intermittent faults can be handled by the alarm handling and recovery control functions, and need not be considered in the context of replicated computations.

5 The Basic Replication Tools

The replication scheme can be seen as a set of tools. The first one of these tools is discussed in this chapter. Before we describe the tools, we will discuss the goal of these tools formally. The ideal, how things should be, is expressed by the replication model. The replication scheme is a more or less faithful implementation of the replication model.

5.1 The Replication Model

The goal of the replication tools is to support replicated computations in a distributed (switching) system with a low level of redundancy. The replicated computations are a prerequisite for implementing graceful unit *changeovers* with an operator command and in the case of the failure of an active unit in 2*N* replicated computers. A *replicated computation is* a pair of *equivalent or consistent computations* which run simultaneously, one in the active unit and the other in the spare unit. Furthermore the *states of the active and the spare computations are consistent*. By the global state of a computation we mean the configuration *C(n)*, where $n \in N$, of the ACT of the respective execution group. A changeover can be visualised as a special event linking the active and the spare ACTs such that after that event the trees are isomorphic. In this section we will formalise these notions.

Notation 5.1 The following symbols are used:

1. $\Sigma^{wo} = \{s_i^{wo} \mid i = 1,...,n\}$ is the execution group of active processes.
2. $\Sigma^{sp} = \{s_i^{sp} \mid i = 1,...,n\}$ is the execution group of spare processes, such that $\Sigma^{wo} \simeq \Sigma^{sp}$.
3. A pair of corresponding or replicate processes of the system: $s = (s^{wo}, s^{sp})$, $s^{wo} \simeq s^{sp}$, $s^{wo} \in \Sigma^{wo}$ and $s^{sp} \in \Sigma^{sp}$, s^{wo} and s^{sp} are of the same type and s^{wo} is executed in an active computer and s^{sp} in its spare.
4. A pair of corresponding or replicate processes of the environment: $\eta = (\eta^{wo}, \eta^{sp})$, $\eta^{wo} \simeq \eta^{sp}$, $\eta^{wo} \in E^{wo}$ and $\eta^{sp} \in E^{sp}$ and η^{wo} and η^{sp} are of the same type.
5. By dest(a) = π, $a \in A$, $\pi = (\pi^{wo}, \pi^{sp}) \in \Pi$ we denote that message a is addressed to both the active (π^{wo}) and the spare process (π^{sp}). By sender(a) = π, $a \in A$, $\pi = (\pi^{wo}, \pi^{sp}) \in \Pi$ we denote that message a is sent by both the active (π^{wo}) and the spare process (π^{sp}) in the same context.

Isomorphic states of the corresponding active and spare processes are not necessarily, and in the case of e.g. the DX 200 system not even usually, bitwise identical because memory allocation for segments is done by each computer unit independently. However, the application does not need to concern itself with these differences between units. An example of isomorphic states is the initial states of two corresponding processes working on the same object (e.g. a call) in the active and the spare computer units.

Definition 5.2 Let $T^{\mathrm{wo}} = (N^{\mathrm{wo}}, \Lambda^{\mathrm{wo}})$ and $T^{\mathrm{sp}} = (N^{\mathrm{sp}}, \Lambda^{\mathrm{sp}})$ be the ACTs for two isomorphic systems $(\Sigma^{\mathrm{wo}}, E^{\mathrm{wo}}, D^{\mathrm{wo}})$ and $(\Sigma^{\mathrm{sp}}, E^{\mathrm{sp}}, D^{\mathrm{sp}})$ where Σ^{wo} and Σ^{sp} are the active and the spare execution groups. The active and spare computations are *equivalent*, iff:

1. the behaviour of the execution group Σ^{wo} is isomorphic with the behaviour of the execution group Σ^{sp}

2. $\forall\, n^{\mathrm{wo}} \in N^{\mathrm{wo}}$ and $n^{\mathrm{sp}} \in N^{\mathrm{sp}}$ such that the configurations $C^{\mathrm{wo}}(n^{\mathrm{wo}}) \simeq C^{\mathrm{sp}}(n^{\mathrm{sp}})$ and $\forall\, a \in A$ if the next node of the ACT of the active system according to Definitions 4.8 is $\grave{\mathrm{n}}^{\mathrm{wo}}$ such that

   $\lambda^{\mathrm{wo}}(n^{\mathrm{wo}}, \grave{\mathrm{n}}^{\mathrm{wo}}) = a$, then $C^{\mathrm{wo}}(\grave{\mathrm{n}}^{\mathrm{wo}}) \simeq C^{\mathrm{sp}}(\grave{\mathrm{n}}^{\mathrm{sp}})$ and $\lambda^{\mathrm{sp}}(n^{\mathrm{sp}}, \grave{\mathrm{n}}^{\mathrm{sp}}) = a$ where $\grave{\mathrm{n}}^{\mathrm{sp}}$ is the next node of the spare system, or

   $\lambda^{\mathrm{wo}}(n^{\mathrm{wo}}, \grave{\mathrm{n}}^{\mathrm{wo}}) = \tau$, then $C^{\mathrm{wo}}(\grave{\mathrm{n}}^{\mathrm{wo}}) \simeq C^{\mathrm{sp}}(\grave{\mathrm{n}}^{\mathrm{sp}})$ and $\lambda^{\mathrm{sp}}(n^{\mathrm{sp}}, \grave{\mathrm{n}}^{\mathrm{sp}}) = \tau$ where $\grave{\mathrm{n}}^{\mathrm{sp}}$ is the next node of the spare system.

It should be noted that because of the Pid allocation scheme, described in Section 1.4.1, Definition 5.2 implies that the members of each of the process pairs are of the same process type. Definition 5.2 also states what an *ideally* replicated computation should be like. However, as we have discussed earlier, we are not going to present an ideal replication scheme but intend to build the scheme based on eventual convergence. That is why we will define a replicated computation more loosely than Definition 5.2, taking into account the requirements of Section 2.4. One of the key requirements in that section was to retain the possibility of using messages which are internal to a computer unit.

Definition 5.3 Let $T = (T^{\mathrm{wo}}, T^{\mathrm{sp}})$, $T^{\mathrm{wo}} = (N^{\mathrm{wo}}, \Lambda^{\mathrm{wo}})$, $T^{\mathrm{sp}} = (N^{\mathrm{sp}}, \Lambda^{\mathrm{sp}})$ be a pair of ACTs representing, in a given environment, all possible behaviours of a pair of the active and the spare execution groups $\Sigma = (\Sigma^{\mathrm{wo}}, \Sigma^{\mathrm{sp}})$ of the system with the nodes $n^{\mathrm{wo}} \in N^{\mathrm{wo}}$ and $n^{\mathrm{sp}} \in N^{\mathrm{sp}}$. The pair of ACTs represent all possible *replicated computations* of the system in a given environment iff for system Σ and for the external messages $a \in A$ between Σ and the environment $E = (E^{\mathrm{wo}}, E^{\mathrm{sp}})$ the following conditions apply:

1. $\exists\, (n_0^{\mathrm{wo}}, n_0^{\mathrm{sp}})$, $n_0^{\mathrm{wo}} \in N^{\mathrm{wo}}$, $n_0^{\mathrm{sp}} \in N^{\mathrm{sp}}$ such that: $C^{\mathrm{wo}}(n_0^{\mathrm{wo}}) \simeq C^{\mathrm{sp}}(n_0^{\mathrm{sp}})$,

2. " a such that <u>sender</u>(a) = η, $\eta \in E$: if a is sent to the system, then

   <u>dest</u>(a) = $(\mathrm{s}_i^{\mathrm{wo}}, \mathrm{s}_i^{\mathrm{sp}}) \in \Sigma$, and " $a \in C^{\mathrm{wo}}{}_Q(n^{\mathrm{wo}})(\mathrm{s}^{\mathrm{wo}})$ where $n^{\mathrm{wo}} \in N^{\mathrm{wo}}$ and " $a \in C^{\mathrm{sp}}{}_Q(n^{\mathrm{sp}})(\mathrm{s}^{\mathrm{sp}})$ where $n^{\mathrm{sp}} \in N^{\mathrm{sp}}$: <u>sender</u>(a) = $\eta^{\mathrm{wo}} \in E^{\mathrm{wo}}$,

3. " a such that <u>sender</u>(a) =s, s $\in \Sigma$: if a is sent to the environment, then

   <u>dest</u>(a) = $(\eta_i^{\mathrm{wo}}, \eta_i^{\mathrm{sp}}) \in E$, and " $a \in C^{\mathrm{wo}}{}_Q(n^{\mathrm{wo}})(\eta^{\mathrm{wo}})$ where $n^{\mathrm{wo}} \in N^{\mathrm{wo}}$ and " $a \in C^{\mathrm{sp}}{}_Q(n^{\mathrm{sp}})(\eta^{\mathrm{sp}})$ where $n^{\mathrm{sp}} \in N^{\mathrm{sp}}$: <u>sender</u>(a) = $\mathrm{s}^{\mathrm{wo}} \in \Sigma^{\mathrm{wo}}$.

So, a replicated computation is *a pair of initially equivalent computations* and the incoming external messages are passed to the two execution groups simultaneously. One of the member computations is executed by the active unit and the other by the spare unit. The messages seem to come from the active environment and the spare environment is hidden. Looking from the environment, the messages sent by the replicated execution group of the system seem to come from an active process and the corresponding spare sender is hidden.

The executions have to be simultaneous because the goal is the possibility of a graceful unit changeover in a real-time environment.

In Definition 5.3 we assumed that $E = (E^{wo}, E^{sp})$ for symmetry with the system because both system Σ and environment E are executed by a computer unit that may or may not be duplicated at any given point in time. We will come back to the dynamic behaviour of a computer unit after a definition and a lemma in this chapter.

Definition 5.4 Let T^{wo} and T^{sp} be ACTs representing the behaviours of the active and the spare members of a replicated system. The global states $C^{wo}(n_1)$ and $C^{sp}(n_2)$ of the replicated system are consistent, iff

a. $C^{wo}({n^{wo}}_1) \simeq C^{sp}({n^{sp}}_2)$, or

b. there was a pair of isomorphic states ($C^{wo}(n'_1) \simeq C^{sp}(n'_2)$) such that

   i. $C^{wo}(n_1)$ was reached from $C^{wo}(n'_1)$ and $C^{sp}(n_2)$ was reached from $C^{sp}(n'_2)$, and

   ii. there exists a pair of isomorphic states ($C^{wo}(n''_1)$ and $C^{sp}(n''_2)$) such that if $C^{wo}(n''_1)$ will be reached from $C^{wo}(n_1)$, then $C^{sp}(n''_2)$ will eventually be reached from $C^{sp}(n_2)$.

Note that according to Definition 5.3 the condition (b) sub-item (i) of Definition 5.4 is always true for the members of a replicated computation. The Definition 5.3 does not guarantee that sub-item (ii) will be true. However, for replicated computations Lemma 5.5 applies.

Lemma 5.5 Let $\Sigma = (\Sigma^{wo}, \Sigma^{sp})$ be a pair of the active and the spare execution groups of deterministic processes of the same type and let Σ^{wo} and Σ^{sp} be reactive[9]. The states of a replicated computation of Σ are consistent, if

1. the execution groups are closed and the active and spare data environments are isomorphic, i.e.

   $\forall$ *s*, *a*, and $(s^{wo}, s^{sp}) \in \Sigma$ the values of the data units such that $(d^{wo}_{s\times a}, u^{wo}_{s\times a}) \in \underline{\text{read}}\, {}^{\sigma^{wo}}_{s\times a}$ are equal to the values of the isomorphic data units such that $(d^{sp}_{s\times a}, u^{sp}_{s\times a}) \in \underline{\text{read}}\, {}^{\sigma^{sp}}_{s\times a}$, and

2. " $\sigma \in \Sigma$ and $a \in \underline{\text{input}}(\underline{\text{current}}(s))$: The order of messages in $C_Q(n^{wo})(s^{wo})$ and $C_Q(n^{sp})($ $s^{sp})$ is the same, and

Proof*:* Here, as before we assume that processes are well-behaved and that the states of a process between receptions of messages are indistinguishable.

Condition b, sub-item (i) of Definition 5.4 is true according to Definition 5.3, item (1).

Condition (2) of Lemma 5.5 will guarantee that messages are received in the same order by the active and the spare processes $\sigma \in \Sigma$ of the member execution groups. This will make the member computations satisfy condition b sub-item (ii) of Definition 5.4.

Consequently, Lemma 5.5 is true according to Definition 5.4 item b.

[9] See Definition 4.13.

The condition that the active and the spare execution groups are reactive means that any computation they may execute is finite unless new messages keep arriving from the environment. Having executed a finite number of steps, they arrive in a steady state which is isomorphic, provided that the rest of the conditions of the lemma are satisfied. We will not take the condition that the active and spare execution groups are closed as a general assumption because if a process of the environment writes into the data units of the execution group, it is possible that the same write action takes place in both the active and the spare computer and thus the states of the execution groups will still remain isomorphic. The definition of a replicated computation could be generalised for an arbitrary number of spares, but in a switching environment this is not of practical interest, and therefore here and in all the rest of this thesis, we will assume that the maximum is one spare computation.

State changes may occur in a system which may be replicated. The possible states and state changes of a replicated system (e.g. a computer unit or a replicated execution group) are presented in Figure 5.1.

The state changes indicated in Figure 5.1 are:

(1) Changeover: the previously active system becomes the spare and the spare becomes active.

(2) Creation of the hot-standby spare.

(3) Deletion of the hot-standby spare.

In our further presentation we will assume that such state changes are possible and will discuss their impact as a separate issue apart from the basic constructs of the replication model and the replication scheme.

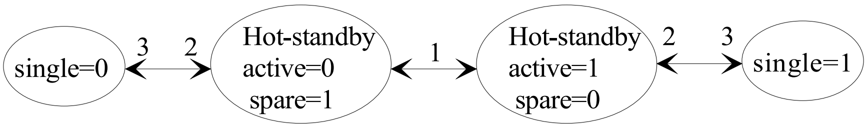


Figure 5.1 States and state changes of a replicated system.

An ideal, thoroughly reliable synchronisation scheme of WO and SP processes is one which guarantees that the member computations of a replicated computation will remain equivalent and thus their states remain consistent. Such a synchronisation scheme of WO and SP processes would require that all events that affect the global state of the process are synchronised between the WO and the SP. The synchronisation should ensure that *equivalent processes are allocated* for a service and that all the events are handled by the WO and SP processes *exactly in the same order*. This order should not be violated by a changeover nor during the creation of the hot-standby spare system. These events include the following:

- all incoming messages: external including time-out messages and internal messages,
- all software resource allocations: hand processes, buffers,
- read actions,
- write actions.

5.2 The Basic Replication Scheme

A Replication Scheme refers to an outline of how replicated computations are implemented in a system. We will call our basic scheme *loose message synchronous mode.* It comprises five elements, two of which conform to the ideas of modular redundancy, two others to the ideas of primary/standby redundancy, and the last is the time-out synchronisation protocol. The intention of the basic scheme is not to meet the requirements of a thoroughly reliable scheme but rather to solve only the key synchronisation problems.

5.2.1 The replication group

The system is divided into a set of replication groups. The replication groups are sets of processes. A hand process may be a member of exactly one replication group. The position of the master processes will be discussed in Section 5.3.

Definition 5.6 The replication group is a pair of isomorphic execution groups Γ^{wo} and Γ^{sp} for which Definition 5.3 holds.

One of the execution groups resides in the WO computer and the other in the SP computer. According to Definition 4.12 of an execution group, processes of each of the execution groups communicate with each other directly with internal messages, independent of the communication which is taking place between the processes of the other group.

More exactly: Let $T = (N, \Lambda)$ be an ACT for a system of processes $\Gamma^{wo} \subseteq \Sigma^{wo}$ and let $\sigma \in \Gamma^{wo}$. Then the set Γ^{wo} is an execution group and the following applies:

1. Γ^{wo} has only one member $s \in \Gamma^{wo}$, or

2. $\Gamma^{wo} = \Gamma' \cup \{s\}$, where $s \notin \Gamma'$ such that Γ' is an execution group and
   $\exists\, s' \in \Gamma'$ such that $\exists\, n, n' \in N$ and $a \in A$: $l(n, n') = a$ and
   i. <u>sender</u>(a) = s and <u>dest</u>(a) = s' , or
   ii. <u>sender</u>(a) = s' and <u>dest</u>(a) = s.

The same applies for $\Gamma^{sp} \subseteq \Sigma^{sp}$.

The execution groups are isomorphic sets of processes. We will call the execution groups the active and spare *members of the replication group*. The internal messages here are also internal to the hosting computer. Membership in the replication group is intended to be static, i.e. for the whole lifetime of a process. The service provider allocation protocol explains how a replication group can be extended by new processes. The replication group concept in the scheme is a modular redundancy element.

Figure 5.2 illustrates the concept of the replication group. Circular figures σ_1, σ_2, σ_3 and σ_4 indicate replicated processes. By arrows the initial user-provider relations are shown. The boxes around the groups of processes indicate the boundaries of the members of the replication group.

A replication group is an instance of a *replication class*. A replication class is a pair of identical sets of process types. In a switching environment N-version programming is typically not used, and thus the replication class member sets have the *same code*[10], and ideally execute their code irrespective of where they reside, in a WO computer, or in a SP computer.

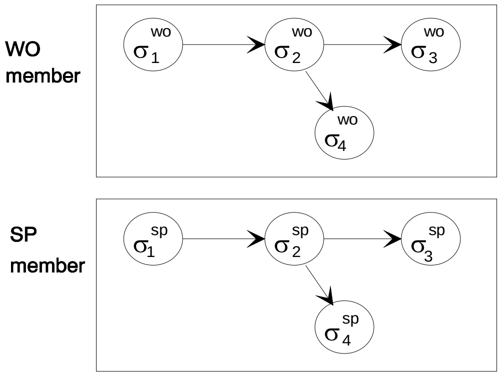


Figure 5.2 An example of a replication group.

5.2.2 The multicast delivery protocol

Another element of the basic replication tool, conforming to the idea of modular redundancy, is the *wo+sp multicast delivery.* The multicast delivery passes a message simultaneously to the members of a pair of corresponding processes of a replication group. This is illustrated in Figure 5.3. Wo+sp is one of the values of the destination delivery code which is part of the computer_address. Thus the multicast delivery addresses items (2) and (3) of Definition 5.3.

Consequently, if $\Gamma_1 = (\Gamma_1^{\text{wo}}, \Gamma_1^{\text{sp}})$ and $\Gamma_2 = (\Gamma_2^{\text{wo}}, \Gamma_2^{\text{sp}})$ are two replication groups, such that A are the messages between them, and $T^{\text{wo}} = (N^{\text{wo}}, \Lambda^{\text{wo}})$ and $T^{\text{sp}} = (N^{\text{sp}}, \Lambda^{\text{sp}})$ are the pair of two ACTs for Γ_1, then

(i). " $a \in A$ such that $\underline{\text{sender}}(a) = \sigma_2, \sigma_2 \in \Gamma_2$: if a is sent to Γ_1, then

$\underline{\text{dest}}(a) = (\sigma_{1_i}^{\text{wo}}, \sigma_{1_i}^{\text{sp}}) \in \Gamma_1$, and " $a \in C^{\text{wo}}_Q(n^{\text{wo}})(\sigma_1^{\text{wo}})$ where $n^{\text{wo}} \in N^{\text{wo}}$ and " $a \in C^{\text{sp}}_Q(n^{\text{sp}})(\sigma_1^{\text{sp}})$ where $n^{\text{sp}} \in N^{\text{sp}}$: $\underline{\text{sender}}(a) = \sigma_2^{\text{wo}} \in \Gamma_2^{\text{wo}}$,

(ii). " $a \in A$ such that $\underline{\text{sender}}(a) = \sigma_1, \sigma_1 \in \Gamma_1$: if a is sent to Γ_2, then

$\underline{\text{dest}}(a) = (\sigma_{2_i}^{\text{wo}}, \sigma_{2_i}^{\text{sp}}) \in \Gamma_2$, and " $a \in C^{\text{wo}}_Q(n^{\text{wo}})(\sigma_2^{\text{wo}})$ where $n^{\text{wo}} \in N^{\text{wo}}$ and " $a \in C^{\text{sp}}_Q(n^{\text{sp}})(\sigma_2^{\text{sp}})$ where $n^{\text{sp}} \in N^{\text{sp}}$: $\underline{\text{sender}}(a) = \sigma_1^{\text{wo}} \in \Gamma_1^{\text{wo}}$,

Entities Γ_1 and Γ_2 in Figure 5.3 are distinct communicating *replication groups* with the active $(\Gamma_1^{\text{wo}}, \Gamma_2^{\text{wo}})$ and spare members $(\Gamma_1^{\text{sp}}, \Gamma_2^{\text{sp}})$. Messages are passed asynchronously without compulsory acknowledgements. Groups Γ_1 and Γ_2 may reside in the same pair of computers or in different pairs of computers. In either case the message is delivered using the hardware supported multicast facility along the Message Bus. This means that the message is sent by the WO computer only once and read by both of the recipient computers at the same time. The

[10] *Graceful software updating* procedures are of interest in the switching environment, i.e. procedures for updating the switching system software package without service interruptions. During such procedures, replication class members may be different versions of the same program blocks.

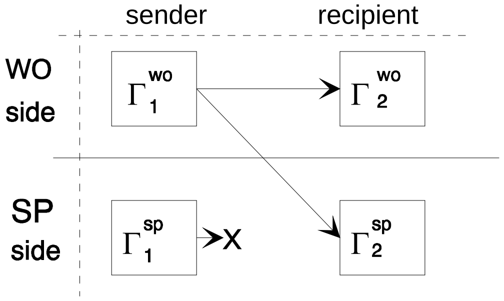


Figure 5.3 Message with wo+sp delivery.

message sent approximately at the same time by the SP member of the replication group Γ_1 is discarded by the kernel. The notation for discarding a message is (→ X) in all the figures. Because all messages with wo+sp delivery are sent through the same bus, such messages are always received by a process in the same order as they were sent by any single process. Furthermore, the sequence of reception of such messages in the active and the spare replicate processes from any number of processes is the same if messages are not lost. This is because, in fact, the same message is received by both the active and the spare computer from a shared communication medium that can provide service only to a single sender at a time.

5.2.3 The service provider allocation protocol

The third element of the scheme is the handling of service provider allocation messages. The protocol is always used for hand allocation inside and across the boundary of a replication group to facilitate the *creation of isomorphic replication group members*. The protocol ensures that isomorphic Pids are reserved for the active and the spare process. In our system this isomorphism is implemented by Pids which are bitwise identical including all the parts of the identifier except for the *destination delivery code* that is a part of the computer address.

To accommodate this new feature we have to extend our model of computations to cover the creation of processes.

Definition 5.7 Let $B \subseteq A$ be the set of the service provider allocation request messages and $\beta \in B$, let Σ be the system of processes s $\in \Sigma$ and E the environment where processes $\eta \in E$ and let Γ be the set of (hand) processes $\gamma \in \Gamma$ which are free and ready to be allocated for a task and included in the system. Let there be a set of master processes M such that $M \subseteq E$, and $\mu \in M$ and let the current state of μ be s_μ. Then T is the ACT for the system (Σ, E, D) incorporating the creation of processes, iff

1. The messages $\beta \in B$ are queued into Q as any other messages.
2. Semantics of the consumption of a message $\beta \in B$ amends item (3) of Definition 4.8 as follows:

   Let $n \in N$, and $\exists\, \gamma \in \Gamma - \text{Domain}(C_S(n))$. Then $\forall\, \mu \in M, M \subseteq E$ such that $\underline{\text{first}}(C_Q(n)(\mu) = \beta$, where $\beta \in \underline{\text{input}}(s_\mu)$: $\exists\, n' \in N$ such that

   a. $C_S(n') = C_S(n) \cup (\gamma, \underline{\text{initial}}(\gamma))$,

   b. $C_Q(n') = C_Q(n)\, \Theta \{(\mu, <\beta>)\} \oplus \{(\text{dest}(b_1), <b_1>)\} \oplus \ldots \oplus \{(\text{dest}(b_n), <b_n>)\} \oplus \{(\text{dest}(a_1), <a_1>)\} \oplus \ldots \oplus \{(\text{dest}(a_m), <a_m>)\}$, where $\underline{\text{output}}(s_\mu, s'_\mu) = <b_1,\ldots,b_n>$, $\underline{\text{output}}(\underline{\text{start}}(\gamma), \underline{\text{initial}}(\gamma)) = <a_1,\ldots,a_m>$, and $s'_\mu \in \underline{\text{next}}(s_\mu, \beta)$,

c. $C_V(n') = C_V(n) - \{(d, u)| \; d \in \underline{\text{writeset}}(s_\mu, s'_\mu) \cap D_\Sigma\} \cup \{(d, u')| \; d \in \underline{\text{writeset}}(s_\mu, s'_\mu) \cap D_\Sigma\} - \{(d, u) \mid d \in \underline{\text{writeset}}(\underline{\text{start}}(\gamma), \underline{\text{initial}}(\gamma))\} \cup \underline{\text{write}}_{\underline{\text{start}}(\gamma)\times\underline{\text{initial}}(\gamma)}$, where u' are the new values written in the transition by the master process to the data units of the system,

d. $\text{l}(n, n') = \beta$.

3. The rest of the ACT is constructed as in Definition 4.8. Item (4) of Definition 4.8 applies to $\beta \notin \underline{\text{input}}(\underline{\text{current}}(\mu))$ and to $\neg\exists\, \gamma \in \Gamma - \text{Domain}(C_S(n))$.

Note that the definition assumes that processes μ and γ send only one message to a destination and thus all the messages in item 3 of the definition are queued in a random order. The definition ignores the order in which μ and γ write into the data units of the system as well as the combined order of write actions and the sending of messages. These assumptions follow the same lines of reasoning as in Chapter 4.

The creation of *replicated hand processes* is modelled by a pair of ACTs T^{wo} and T^{sp} each of which is constructed according to Definition 5.7 with the addition that the definition applies to $M = (M^{\text{wo}}, M^{\text{sp}})$ and $\gamma = (\gamma^{\text{wo}}, \gamma^{\text{sp}}) \in \Gamma = (\Gamma^{\text{wo}}, \Gamma^{\text{sp}})$ and in item 2 of the definition $\gamma^{\text{wo}} \in \Gamma^{\text{wo}} - \text{Domain}(C_S(n^{\text{wo}}))$ and $\gamma^{\text{sp}} \in \Gamma^{\text{sp}} - \text{Domain}(C_S(n^{\text{sp}}))$.

The protocol implementing the allocation of hand processes in a replicated environment is as follows (Figure 5.4):

1. The service provider allocation request is sent by the service user only to the WO master process of the service provider program block. The message sent by the active user is delivered; the message sent by the spare user is discarded.
2. The master process of the service provider invokes a kernel primitive to allocate a hand with the request message as a parameter. The `allocate_hand` primitive selects a free hand using the *Pid search algorithm*, defines the *Pid* of the hand (finding the hand and finding the local Pid is the same; the focus[11] has to be incremented), and sends a kernel message to the spare master. The kernel message carries the allocate request and the local Pid and focus of the hand. The primitive sets a time limit for the acknowledgement. When the kernel message has been sent, γ^{wo} hand exists for other processes of the system.

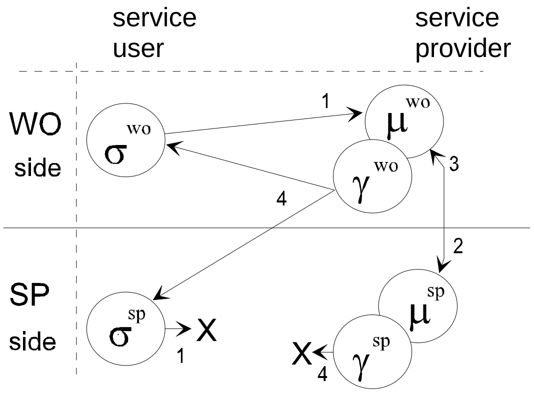


Figure 5.4 Service Provider Allocation Protocol.

3. The spare master allocates a hand with the *isomorphic Pid* and sends an acknowledgement kernel message to the WO master. From this moment we say that γ^{sp} hand exists and is

[11] Indicating the beginning of the next life of the process.

involved in a computation. A *replication state variable*, "Replicated_mode_ok" of the WO and SP hands is set to true. The `allocate_hand` primitive returns in both the active and the spare service providers. The service provider master and the newly allocated hand processes are free to continue as appropriate for the application. When the acknowledgement message has been received by the WO, we say that the γ^{wo} hand is involved in a computation.

The `allocate_hand` primitive, invoked by the spare master, also runs the Pid search algorithm locally. If the result of the search is a different local Pid value than the one sent by the WO, both hands are reserved. The hand suggested by the WO is always allocated for the new task. The locally found hand is set into the *suspended state* to wait for a release message from the WO. This is done because it is possible that this is an error caused by a loss of a message in one of the previous applications of the service provider allocation protocol. By indicating this error, a hook for error correction is created. This will be dealt with in Chapter 7 of this thesis.

If the required hand is not free in the SP unit, which is always regarded as an error situation, it is forcefully released and allocated for the new task.

If the time limit in WO expires first, a kernel message is sent once more[12] and a new time limit is set. If the timer expires the second time, the failure is reported to a supervisory function, and the WO continues irrespective of what happens on the other side. Naturally in this case the "Replicated_mode_ok" of the WO hand is set to false.

4. The usual procedure is that the WO provider hand sends an acknowledgement with wo+sp delivery to the service user. The user gets the identity of the service provider in the acknowledgement message. The corresponding message sent by the spare service provider is discarded.

5.2.4 The hand release protocol

This element of the replication scheme allows destroying replicated processes and consequently replicated computations in a controlled manner. To formalise the semantics of the deletion of processes from the system we will first extend our model of computations.

Definition 5.8 Let $B \subseteq A$ be the set of the release request messages and let $\beta \in B$, let Σ be the system of processes $s \in \Sigma$ and E the environment where processes $\eta \in E$. Then T is the ACT for the system (Σ, E, D) incorporating deletion of processes, iff

1. stop: $\Sigma \rightarrow S$ is a mapping from the set of processes of the system to a state after which nothing can happen[13].
2. The messages $\beta \in B$ are queued into Q as any other messages.
3. Semantics of the consumption of a message $\beta \in B$ amends item (2) of Definition 4.8 as follows:

   Let $n \in N$. Then $\forall$ s $\in \Sigma$ such that $\underline{\text{first}}(C_Q(n)(s)) = \beta$, where $\beta \in B \cap \underline{\text{input}}(\underline{\text{current}}($ s)): $\exists! \, n' \in N$ such that

[12] This retry procedure is not currently implemented in the DX 200 system.

[13] This corresponds to the STOP statement in SDL and TNSDL.

a. $C_S\,(n') = (C_S\,(n);\ \mathrm{s}\!: \underline{\mathrm{stop}}(\mathrm{s}))$, and $C_S\,(n)(\mathrm{s})$ is removed from $C_S\,(n')$,

b. $C_Q\,(n') = C_Q\,(n)\ \Theta\ \{(\mathrm{s}, <\beta>)\} \oplus \{(\mathrm{dest}(b_1), <b_1>)\} \oplus \ldots \oplus \{(\mathrm{dest}(b_n), <b_n>)\}$, where $\underline{\mathrm{output}}(\underline{\mathrm{current}}(\mathrm{s}), \underline{\mathrm{stop}}(\mathrm{s})) = <b_1,\ldots,b_n>$, additionally $C_Q\,(n)(\mathrm{s})$ is removed from $C_Q\,(n')$,

c. $C_V(n') = C_V(n) - \{(d, u) |\ d \in \underline{\mathrm{writeset}}(\underline{\mathrm{current}}(\mathrm{s}), \underline{\mathrm{stop}}(\mathrm{s}))\}$

$$\bigcup \underline{\mathrm{write}}_{\underline{\mathrm{current}}(\sigma)\times\underline{\mathrm{stop}}(\sigma)}\ .$$

d. $\mathrm{l}(n, n') = \beta$.

4. The rest of the ACT is constructed as in Definition 5.7. Item (4) of Definition 4.8 applies to $\beta \notin \underline{\mathrm{input}}(\underline{\mathrm{current}}(\mathrm{s}))$.

Note that the definition assumes that process s sends only one message to a destination and thus all the messages in item 3 of the definition are queued in a random order. The definition ignores the combined order of write actions of s and the sending of messages. These assumptions follow the same lines of reasoning as previously.

The deletion of *replicated hand processes* is modelled by a pair of ACTs T^{wo} and T^{sp} each of which is constructed according to Definition 5.9 with the addition that

$\underline{\mathrm{stop}}\!: \Sigma \to S$ is defined for $\Sigma = (\Sigma^{\mathrm{wo}}, \Sigma^{\mathrm{sp}})$.

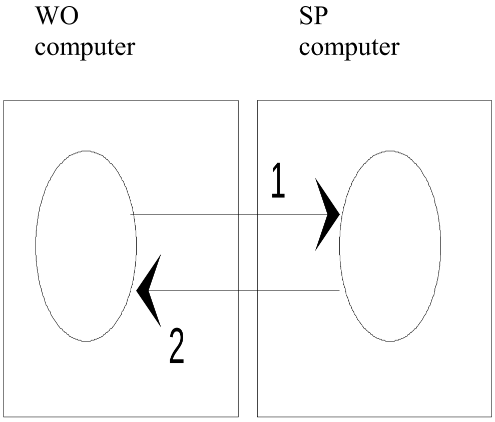


Figure 5.5 The Hand Release protocol.

The hand release protocol anticipates the possible problems arising if some hands would be free in the spare, while their twins in the active unit were allocated to a task. Such problems could arise e.g. after a unit changeover. This protocol ensures that inconsistent processes in the spare are easily distinguished from consistent and from free processes and thus creates a possibility for further development of the replication scheme. The protocol is invoked by a hand upon completion of its task by calling the `main_receive` kernel procedure with the *release-me* parameter set to true. This hides the details from the application. The protocol is as follows (Figure 5.5):

1. In the WO unit the `main_receive` primitive sends a release message to the SP, sets a time limit and waits for an acknowledgement.

2a. In the SP, the `release` primitive checks whether the release message has already arrived. If yes, it inserts the hand into the free hand queue, sends the acknowledgement to the WO and ends. If the release message has not arrived yet, the hand is left in a *suspended state* to wait for it. The suspended state can be used for error detection, namely if it lasts too long, it is highly probable that the computation in the spare is in an inconsistent state. The spare process can legitimately be in the suspended state, if the last transition after which the hand is released has been executed in the spare unit while in the active unit the same transition is about to begin or has not yet been finished executing. Because of Lemma 5.5 and Definition 5.4 we know that eventually − and in practice very soon − the active process should finish the last transition, and according to item (1) of this procedure, send the release message to the spare.

2b. When the release message arrives in SP, and the hand is found in the suspended state, it is inserted into the free hand queue and an acknowledgement is sent to the WO unit.

3. In the WO unit, if the time limit expires[14], the active hand is put into the free hand queue and the kernel passes control to another process.

5.3 State changes of a replication group

As noted earlier in Section 5.1, state changes are possible in our target system (see Figure 5.1). Here we will briefly discuss the impact of such state changes into the elements of the replication scheme.

We will show that the impact of the state changes to the elements of the replication scheme is temporary, provided that the measures identified underneath are taken.

5.3.1 Changeover

In a changeover the members of a replication group change places. After the changeover the former spare member becomes active and thus visible to the environment. This visibility manifests itself through the multicast delivery protocol. Due to the computation skew between the active and the spare computers the multicast delivery protocol may cause a loss or a duplication of a message during a unit changeover of the sending computer. A message will be lost if at the moment of unit changeover the spare was ahead of the active and had just before the changeover sent a multicast message that was discarded in SP. After the changeover the same message will be discarded again in the new spare unit. If the spare was lagging behind at the moment of the changeover of the sending unit, the message will be sent twice, first by the old active and then by the new active unit.

Provided that nothing is done to eliminate the message loss and duplication problem a unit changeover may increase the risk of failure for some of the computations. The problem may cascade back to the sender if the message was lost and the sender waits for an acknowledgement. When the time-out for the acknowledgement elapses, it is up to the application either to retry or to launch an abort or an equivalent procedure. The receiver should be able to discard the second message if a message was sent twice. It should be noted, however, that replication arrangements are not the only reason for such retry, abort and validation procedures in the applications. Because of the low level of redundancy, partial failures cannot be masked in the system and a service user always has to assume that the provider may fail, and make the best decision from the point of view of the application. The implementation of our failure model requires that the validity of received messages can be established. The system provides primitives to support the validation, but the selection of the validation method is left to the application.

[14] For performance reasons there is no retry procedure but internally an error is indicated.

Leaving the retry and message validation responsibility to the application, although it is a burden, has its advantages. When an acknowledgement is received from a peer process, the recipient may be sure that both parties understand the situation alike and e.g. that the peer process is not in an inappropriate state nor in an infinite loop. An acknowledgement on a message protocol level does not carry this information.

To eliminate the problems in the multicast delivery the protocol would have to be synchronised with the changeover or alternatively the replicate processes (i.e. lightweight process families and all normal processes) immediately before the changeover. This can be done by sending a multicast synchronisation message to the replicate processes before the changeover and setting them to wait for the recovery to switch the sides of the active and spare computers. This solution is applicable if the maximal halt period is within the limits determined by the real time requirements discussed in Chapter 2. These kinds of message loss and duplication errors are, however, assumed to be not too frequent nor fatal for call control type of applications and that is why they are ignored in the basic replication scheme.

If a changeover occurred while the service provider allocation protocol was being executed, two consecutive applications of the protocol may come into conflict. Such conflicts may be avoided by synchronising the master processes with the changeovers. The system recovery control function can take care of this by making the changeover known to the master processes which should not allocate new processes while the changeover is in progress.

If the unit changeover occurred between the release message (message (1) in Figure 5.5) sent by the old active unit and before the release procedure is invoked in the new active unit, the new active unit will ignore the earlier release message and send a release message after the changeover. The new spare process, waiting for the acknowledgement to his release message, now has to accept the release message based on the new unit state and proceed as described in the hand release protocol.

After a changeover in the active unit there may be processes in the suspended state. To avoid the creation of an orphan process which will never be released, there must be a time limit for the suspended state. When the time limit expires, such processes will eventually be put into the free hand queue. If the process in the suspended state is in the active unit, the hand release protocol is applied.

5.3.2 Deletion of the hot-standby spare member

The spare system is isolated from the environment by the recovery control that changes the address mapping so that the multicast deliveries are sent only to the remaining active system by the environment. The active system will now bypass the steps in the service provider allocation, hand release, and time-out synchronisation protocols, which assume the existence of the spare.

5.3.3 Creation of the hot-standby spare member

This requires that the state of the spare has to become consistent with the active system. Two approaches tackling this problem are discussed: the passive warm-up in this section and active warm-up in Chapter 6.

When the spare unit is taken from the testing state (TE) to the standby state (SP), the unit is restarted and first set to the SP-UP sub-state to bring it up to the same dynamic state as where the WO unit is. More exactly in the SP-UP state, *by the warm-up procedure, the spare unit is* *taken from the initial steady state to a consistent state with the active unit*. In the warm-up procedure the state oriented processes are involved, while stateless processes have nothing to do with warm-up in the narrow sense. The initial steady state which is at the same time the correct working mode of stateless processes can be reached by loading the appropriate code of the program blocks and data files and starting the master processes.

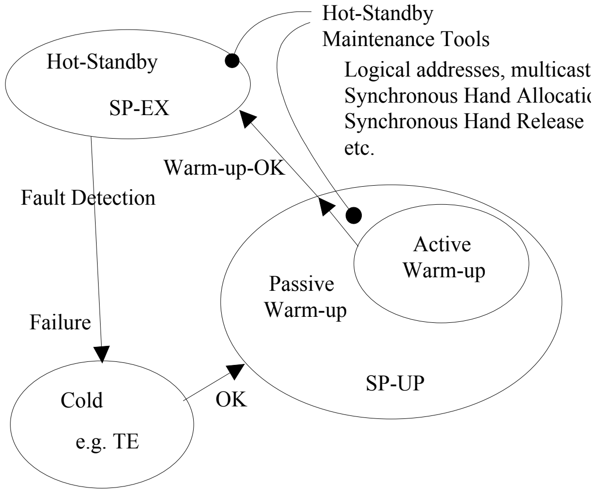


Figure 5.6 Use of Replication Tools in different unit states.

The loose synchronous mode can easily support passive warm-up of the spare unit. By passive warm-up we mean that the new computations are created as replicated computations, and with time the number of consistent computations in the spare unit will get closer and closer to the total number of parallel computations in the WO unit. This scheme works for finite computations, like calls. Figure 5.6 gives an overview of the use of replication tools, like the basic replication tools and warm-up in different states of a functional unit.

Clearly there is no guarantee that the spare unit will ever reach a consistent state with the active i.e. that the passive warm-up will successfully end, nor does the basic scheme provide an end criterion for the warm-up. For this reason and because the passive warm-up might take too long, active warm-up is needed. By the active warm-up we mean the procedure of copying the current values of the state variables of the state oriented processes in the active unit to the corresponding state variables of the spare. Active warm-up also gives a criterion of whether the warm-up has successfully ended.

To support the passive warm-up, certain considerations have to be taken into account in the protocols of the basic replication scheme. The passive warm-up works as follows:

1. The initial state of the spare unit is such that all programs and files have been loaded and the master processes are ready to allocate hand processes and perform possible other tasks.
2. The loose message synchronous mode is started in the two units by state information delivery initiated by the recovery control. The spare unit is declared as SP in all the units of the system in the address mapping tables of the kernel and the message bus interface of the SP-UP unit is programmed to receive the wo+sp multicast messages.
3. From now on all the new computations are created as replicated computations using the loose message synchronous mode protocols. Considerations to be taken into account in the protocols are:
   a. The `allocate_hand` primitive invoked by the spare master should not run the Pid search algorithm locally, because it would give results that should be expected to be

different from those received from the WO unit. The primitive should just allocate the same hand as in the WO unit.

b. If the computation ends and the hands are released using the release protocol, and the hand is already free in SP, this should not be regarded as an error. If the hand was reserved in the SP, it is released using the normal protocol.

c. When a multicast message is received in the SP, and the recipient process is found to be free (not allocated to a task), the message is discarded by the kernel and this is not regarded as an error.

An alternative approach to the above considerations concerning the hand allocation protocol would be to try to synchronise the free hand queues. This is not suggested because to be able to handle items (b) and (c), the kernel has to be aware of the SP-UP state all the same, and being aware of this state, it is simpler to solve the problem as described in item (a).

5.4 Analysis of the properties of the Basic Replication Scheme

The optimistic failure assumption defined in Section 2.4 (Assumption 2.1) in the context of replicated computations means that hardware failures in the active computer are *detected by the software fast enough* for the unit changeover to take place before the failure propagates itself to another computer unit of the system e.g. to the spare computer.

The loose message synchronous mode comprises the basic computation replication tool. It may be used by some of the applications as such. It can, however, easily be seen, that this tool does not guarantee even conditional instantaneous correctness. The advantages of the loose message synchronous mode are:

- It is very simple, and has a very small performance overhead in applications like call processing, especially if replication groups can be kept big enough. We will return to this issue in Chapter 10.
- It can be implemented in a way which makes the replication of computations easy to implement to the application programmers. This topic will be addressed later in this thesis.

The consistency properties of the scheme can be summarised with a state model of a replicated process represented in Figure 5.7. For simplicity, in the model it is assumed that the service provider allocation and hand release protocols do not fail.

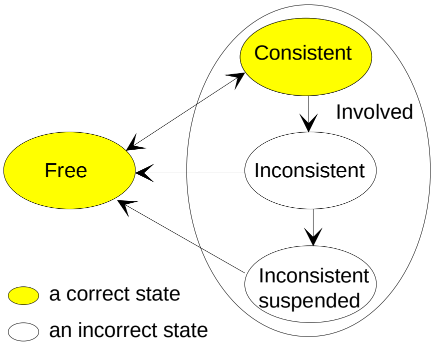


Figure 5.7. State model of a replicated hand process.

In Figure 5.7 the states are as follows. "Free" means that the process pair is not allocated to any task, "Involved" that both the active and the spare process are engaged in a computation. "Involved" has the substates: "Consistent" in which the spare process is in a consistent state with the active, i.e. Definition 5.4 applies, "Inconsistent" - i.e. unknown to the system or to the application Definition 5.4 is not true for the process pair, "Inconsistent suspended" means that the spare process is in the suspended kernel state - i.e. the system can assume, after a time-out, that the spare is inconsistent.

The master processes are not included in the execution groups of a replication group. The idea is that only the hand allocation and release protocols (Sections 5.2.3 and 5.2.4) determine the states of the masters, while all interactions related to the actual work are handled by the hands and may thus influence the state of the hand processes. Likewise the state model of Figure 5.7 does not apply to a master process e.g. because from the point of view of the replication scheme, a master process is never free nor suspended.

Lemma 5.9 Let $\Sigma = (\Sigma^{wo}, \Sigma^{sp})$ be a replication group and let $s^{wo} \in \Sigma^{wo}$ and $s^{sp} \in \Sigma^{sp}$. Let us assume that the service provider allocation and hand release protocols always succeed. Then

1. $\forall\ s^{sp} \in \Sigma^{sp}\ \exists\ s^{wo} \in \Sigma^{wo}$ such that $s^{wo} \simeq s^{sp}$, we call this the *existence property*,

2. $\forall\ s^{wo} \in \Sigma^{wo}$: $\underline{\text{Involved}}(s^{wo}) \Rightarrow \underline{\text{Involved}}(s^{sp})$ and $(s^{wo} \simeq s^{sp})$, we call this the *non-contradictory computations* property,

   where Involved: $\Sigma \rightarrow \{\text{true, false}\}$ is a Boolean function the values of which are true for processes, which are not free, the allocation acknowledgement from the SP has been received, and the hand release request has not been invoked in the WO. A process in the suspended state is involved in the computation it was executing when it was set to this state.

Proof: Three cases have to be studied:

a. Let us first assume that no unit changeovers of the computer executing Σ have occurred. Item (1) holds, because according to the hand allocation protocol s^{wo} is allocated first from the queue of free hands on the WO side and in the SP unit an isomorphic s^{sp} is always reserved for the computation. Further, if the spare falls into an inconsistent state and even is suspended, it is always released according to the hand release protocol at the same time as the active process.

   Implication (2) holds if s^{wo} is free. Let $\underline{\text{Involved}}(s^{wo})$ = true. According to the hand allocation protocol the isomorphic spare process s^{sp} becomes involved earlier than the active. Even if the spare becomes inconsistent or suspended, still it stays involved in the same computation. Consequently, the implication is true.

b. Let us assume that a unit changeover has occurred. All old s^{wo} become new s^{sp} and old s ^{sp} become new s^{wo}. During a unit changeover no new processes are allocated. Consequently, according to item (a) the lemma holds for new replication groups created after the unit changeover. In Section 5.3.1 we also saw that the hand release protocol will not fail due to a unit changeover occurring in the middle of the protocol.

c. After the unit changeover, items (1) and (2) hold for s^{wo} and s^{sp} which are consistent. According to the state model of a replicated hand (Figure 5.7) a new s^{wo} may also be 'inconsistent' or even suspended. This does not violate item (1) nor item (2). As we said in Section 5.3.1 there is a time limit for the suspended state. The hand release protocol applied to such processes after the time limit of the suspended state expires makes sure that both processes are released at the same time and (1) and (2) remain intact.

The tool guarantees that concurrent computations are not mixed-up. So, if copies of the same code are executed on both sides and if the service provider allocation requests and the acknowledgements of the provider are passed successfully, the scheme guarantees correct creation of isomorphic members of the replication group under our model of failures. I.e., *if the creation of the two members of the replication group is successful, then they are guaranteed to be isomorphic and initially consistent*. Furthermore both of the corresponding active and spare processes remain involved in the same computation until they are released by the hand release protocol at the same time.

The scheme guarantees that if the spare computation ends up in an inconsistent state, an involved spare hand process cannot be confused with a free hand process. The hand release protocol, being successful, guarantees that a spare process is released even if it is in an inconsistent state. This means that processes in a pair of computers with isomorphic *Pids*, can not be involved in different computations.

Lemma 5.10 Let $\Sigma = (\Sigma^{wo}, \Sigma^{sp})$ be a closed replication group, let $s^{wo} \in \Sigma^{wo}$ and $s^{sp} \in \Sigma^{sp}$, let $E = (E^{wo}, E^{sp})$ be the environment, $\eta^{wo} \in E^{wo}$, $\eta^{sp} \in E^{sp}$, and let $\forall\ s, a$, s: $\underline{\text{readset}}^{s^{sp}}(s, a) \cap D^{sp}_{E} = \varnothing$. Then if the states of the processes of the environment (η^{wo}, η^{sp}) become inconsistent, the effects caused by this will not propagate into Σ.

Proof: Let $T^{sp} = (N^{sp}, \Lambda^{sp})$ be the ACT for $(\Sigma^{sp}, E^{sp}, D^{sp})$. Item (3) of Definition 4.8 will not change the global state of the ACT because the replication group is closed. Consequently, the next global states of the ACT are determined by item (2) of Definition 4.8 and the hand allocation and release protocols. All the messages consumed by Σ^{sp} are received from E^{wo} and the read actions upon reception of messages from E^{wo} cannot access possibly inconsistent data in D^{sp}_{E} because of the conditions of the lemma. The applications of the hand allocation and release protocols in Σ are not dependent on E^{sp}. Consequently, the lemma follows.

5.4.1 Analysis of possible errors

The message loss and duplication error type which is inherently possible due to the simple multicast protocol e.g. in conjunction with unit changeovers was discussed in Section 5.3. Error propagation will be further discussed in Chapter 7. Additionally, the following should be considered:

1. If a multicast delivery message correctly arrives in the WO computer but not in the SP computer, there is nothing in the scheme itself that would help to keep the spare computation in a consistent state. It is expected to be typical that the spare process would release itself through a time-out procedure and would end up in the suspended state. If a multicast message arrives in the SP, but not in the WO, the computation in the SP goes ahead of the WO computation. This will not, however, propagate inconsistency outside the replication group by message passing, because all the external messages sent by the spare execution group of the replication group are discarded. There are two possible outcomes for the WO computation in such a situation. Either it is successfully restored by a retry procedure in the application, or it is aborted through a time-out procedure. If the retry is successful, the computations in WO and SP may or may not be resynchronised. This is not guaranteed by the replication scheme, but depends on the application. In case the WO computation is aborted, the release protocol will eventually fix the problem if the hands are released by the application.

2. Replicate computations in the two units may also fall into an inconsistent state because of concurrent messages. The Message Bus guarantees that the order of external messages as seen by both members of a replication group, is the same. However, in no way does the scheme guarantee, that from the point of view of a pair of processes in the different members of a replication group, the combined order of internal and external messages will remain the same in both members of the pair.

   In a switching environment e.g. in call processing, these concurrency situations are certainly possible but statistically not too frequent. The most obvious example is the clearing phase of the call, in which the switching system will be prepared to act concurrently towards both subscribers and the clearing can be initiated by any of them at any time during the call. Fortunately in this case the hand release protocol will eventually resynchronise the WO and SP unit states. Another example is a time-out situation, when a replicated process is waiting for an internal message. The order of the time-out and the message may be different in the WO and the SP computers. The probability of these concurrency situations is kept small by making sure that the time-outs are long enough and by the time-out synchronisation protocol which will be presented in Section 5.5.1.

   The property of restricted error propagation of Lemma 5.10 is equally valid for these kinds of errors. Consequently, we can see that with the replication group concept there is a *trade-off between reliability and performance*. Replication groups which are larger than one process provide the benefit of increased performance due to messages that are internal to a computer unit. The cost for this is an additional possibility of losing consistency between the active and the spare process. So, the replication group concept can be seen as a performance optimisation tool.

3. File read and write actions are not synchronised. An active and a spare process may read different data, if a write happened before the read in one of the computers and is late in the other. These errors happen e.g., when multiple copies of a file are maintained for performance reasons to deliver some data for general use. Also, if a file is written by a replication group and read by another, *inconsistency may propagate* from the first group to the other *through the file*. Secure ways of using memory resident files can be pointed out. They include a file or a data unit which is accessed only by one process, a producer process and a consumer process with synchronised access to a file, and accessing data only through the memory resident database which has the responsibility of maintaining data consistency.

4. If a spare computation is in an inconsistent state, when the unit changeover takes place, the computation will fail. When the model is applied to the call control, it is assumed that the

call will be mishandled. It is assumed that this is acceptable as long as the probability is low enough. An important feature of the scheme is that if the hand processes were allocated for a limited period of time, they will eventually be released and the *normal overall system service capability will be recovered.*

5. Long transitions due to disabled interrupts or high priority and cascading of high priority transitions may cause the hand allocation and release protocols to fail. However, replication is not the ultimate reason why, disabling interrupts for a long time, long high priority transitions and bursts of high priority transitions are troublesome. The basic real time and delay performance requirements, discussed in Section 2.3.1., should not be forgotten.
6. Ignoring the state of the free Pid queues, the states of the active and the spare computer are consistent, except for those replication groups in which a synchronisation error has occurred. Possible cases of synchronisation errors were identified in items (1), (2), and (3). The states of the free hand queues were discussed in Section 5.3.3.

To summarise: Because of errors discussed in items (1), (2), (3) and (4), and in Section 5.3.1, the scheme does not guarantee that the behaviour of replicate processes will remain consistent for the lifetime of a computation. However, due to Lemmas 5.9 and 5.10 errors will remain local over parallel computations in a computer and with respect to dependent computations in a set of computers. We will come back to error propagation issues in Chapter 7. Because of the described non-recoverable errors with low, but non-zero occurrence probability, the behaviour of a replicated computation using this tool, can be conveniently characterised by a reliability model. This is done in Chapter 8.

The scheme is also open to enhancements on the basis of error detection and additional synchronisation points during the execution. This topic will be further discussed in Chapter 7.

The scheme can easily be applied to indefinite length computations. Its application to permanent computations is somewhat problematical, because there is a non-zero probability of losing consistency unrecoverably at a given time interval. This means that the consistency will be lost with a probability, which grows with time and can be made arbitrarily close to certainty, if an arbitrarily long time interval can be taken.

5.5 Implementation Issues

This basic replication scheme has been implemented in the DX 200 system. When making implementation decisions, much concern was given to high performance. In particular, this meant that execution times of such primitives as sending a message had to be minimised.

The implementation of the loose message synchronous mode in the DMX is transparent to an application, except for the service provider allocation protocol, the use of which requires a slightly different program code in the service provider masters. The hand code remains fully immune to the basic replication problems. We will come back to the design rules and recommendations which are introduced due to the replication tools. Also an alternative allocation scenario has been implemented, the difference being that the hand allocation request is sent with the wo+sp delivery code and received on both sides. If TNSDL is used to write a master process, this scenario is transparent to the code of the master.

5.5.1 Time-out synchronisation

Considering error type (2) created by concurrent messages in Section 5.4.1 we can improve the scheme by synchronisation of the time-outs. The time-out messages are important because of their wide use in a real-time system. The clock tick and computation skew between the WO and the SP computers can cause an inconsistent state of the computation to occur if a process receives a message at the end of the time limit during the critical period associated with the skew.

DMX kernel provides real-time services to the applications. These services are:

- time-outs. Time-outs are queued by the System Master kernel process in the order of expiration. The queue is located in the data segment of the System Master process.
- wake-up services.
- time measurement records; these records may be located in any application data segment. They are not used for supervision purposes.
- clock-time services.

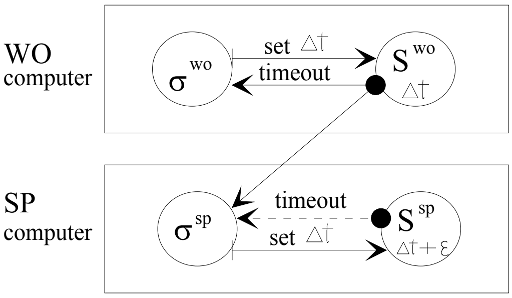


Figure 5.8 Time-out synchronisation protocol.

Legend:

σ	a replicated application process
S	System Master process
set Δt	time-out request procedure call
timeout	time-out elapsed message
ε	a small time interval

The clock tick interval of the System Master is 10 ms. This is also the accuracy of the time-out and wake-up services. When a time-out is set, the WO and the SP units may have an execution skew and so the time-out is not set at exactly the same time on both computers. When a time-out of a replicated process expires, the time-out message may arrive 10 ms plus the execution skew later on one of the sides than it arrived on the other side. Another message arriving during this period could cause an inconsistent state of the replicated computation to occur. To minimise these problems a time-out synchronisation protocol is used. Figure 5.8 gives an overview of the protocol.

Real time services in a switching environment are a very performance sensitive area because in a computer unit thousands of time-outs may be active at the same time. It is expected that most of the time-outs are not intended to elapse in the normal case but are set to recover the application in a situation where an expected message does not arrive. That is why a light protocol was chosen with no synchronisation overhead at the time-out setting.

The protocol goes as follows:

1. Both the WO and the SP set a time-out of Δt with a procedure call. On the SP side, System Master which is the time service provider automatically adds a small value ε to the requested time to make it highly improbable that the time-out would elapse earlier on the SP side.
2. The time-out expires in the WO unit. A time-out message is sent to the application process pair σ with wo+sp delivery[15]. When the time-out is received by the σ^{sp} , the kernel finds the time-out in the incoming message queue of σ^{sp} or in the time-out queue of S^{sp}; and deletes it.
3. If the spare did not receive the time-out from the active unit, at Δt+ε the time-out message is sent by S^{sp} to σ^{sp}.

Some qualitative properties of the protocol are:

1. If the multicast time-out message to the SP is lost and both computations are running, the internal time-out message is sent on the SP side at $\Delta t + \Delta c + \Delta r + \varepsilon$, where Δt is the time interval in the time-out request, Δc is the clock tick skew (less than or equal to half a tick), which is assumed to be constant during Δt, Δr is the computation skew at the time when the time-out was set, and ε is the value added to the time-out interval. The idea is that ε, which is a multiple of the clock tick interval, would compensate for Δc+Δr, and thus make the probability of

   $$P\{ \Delta c + \Delta r + 1 < \varepsilon \} \text{ , where one stands for a clock tick interval,} \tag{5}$$

   close to certainty, but still not violate the intention of the application process noticeably. Thus the idea is to maximise the probability of the time-out message being sent later in the SP computer than in the WO computer. Another way to look at ε is to say that it is the time limit for the time-out message from the S^{wo}.

   Reasoning which justifies the need for the additional tick in Eq. 5 is that even if the computation skew is close to zero and Δc is smaller than the computation skew, it may happen that the setting of the time-out in the spare unit falls into the previous tick interval compared to the active computer. Then the time-out message without ε would be sent in the SP by $1 + |\Delta c|$ earlier than in the active unit. Then there is a risk that $\varepsilon = 1$ is not enough to make sure that the spare time-out will elapse later than the active unit time-out.
2. If the SP computation has gone ahead of the WO computation more than ε - Δc at the time when the time-out is set, the time-out will expire first on the SP side.
3. When the time-out message is received in the spare, the time-out data can always be found, even if it is already in the incoming message queue of the requesting process, and it can be deleted. This avoids the possibility that a time-out would elapse twice
4. Time-outs which were set before a unit changeover and would elapse after it, will typically expire first in the spare system. This is the exact situation which we wanted to avoid. To

[15] In the earlier implementation the multicast was not used, but a synchronisation message was sent to the spare System Master, which sent the time-out to the spare application. Sometimes this caused problems, because the synchronisation took too long.

overcome the problem the system master should adjust the time-outs during the changeover.[16]

The time measurement services do not require any additional synchronisation tools. The clock-time synchronisation problems have been discussed in Chapter 2.

5.5.2 Logical address management

The multicast delivery was implemented in the DX 200 using a *logical addressing* scheme for computer units. These logical addresses are part of the global process identifier, as we saw in Section 1.4. They can be used for addressing messages along with physical computer addresses. A logical or physical computer address is part of every message header. A logical computer address basically has two parts: the *logical unit number* and the *(destination) delivery code* or the range of the delivery. In DMX the message is sent to a destination address, but the recipient automatically sees the address of the sender in the address fields of the message.

A logical computer address is mapped to a physical address of the receiving computer on the Message Bus by the `send` primitive of the kernel. The message bus interface of a computer unit is able to recognise four addresses: the individual unit address, two freely programmable addresses and a broadcast address, which is the same for all units. Programming of the message bus addresses is a unit state control mechanism for which the recovery control is responsible. Recovery control is also responsible for delivering a mapping table to every unit. This mapping table is used by the kernel. A row in the table is directly addressed by the logical unit number. The table basically has three columns. One for the WO units, the second for the SP units and the third for the multicast wo+sp delivery. The column in the row is directly chosen by the destination delivery code in the message. "Discard message" is the fourth possible delivery code.

There are also some special logical addresses, like the *own-unit logical address* and the *pair unit logical address*.

Recovery control uses hardware supported broadcast to deliver unit state information to all the control computers simultaneously. At unit changeover the recovery has to change a row in the address mapping table and deliver the change to all the units. Furthermore, the programmable addresses of the active and the spare unit in the process of unit changeover have to be reprogrammed by the recovery.

5.5.3 Message attributes

Another implementation concept which should be briefly addressed here are the *static message attributes*. A message attribute in the DMX is a bit variable carried in the message header, which may radically influence the treatment of the message by the kernel. An example of a message attribute is the *discard-if-not-wo* attribute, which is used by the loose message synchronous mode. In case a message carries this attribute, the message will be discarded, if it was sent by a process in a computer that was not in the WO state. Other computation replication tools also rely on their own message attributes. They will be introduced later.

[16] This has not been implemented in the DX 200 system.

6 Warm-up of Computations

In Section 2.2. the requirement was set that the virtual machine should provide for the *creation of a hot stand-by spare process* for an active one. On a computer unit level this means that a spare unit is taken from the cold-standby to the hot-standby state. First the unit is restarted and set to the SP-UP (spare-updating) sub-state. In the SP-UP state, by means of the warm-up procedure, the spare unit is taken from the initial steady state to a state consistent with the active unit. In the warm-up procedure only the state oriented processes are involved, while stateless processes have nothing to do with warm-up in this narrow sense. The initial steady state which is at the same time the correct working mode of stateless processes can be reached by loading the appropriate code of the program blocks and data files and starting the master processes.

In Section 5.3.3 we saw that there was no guarantee that the passive warm-up would successfully end. Because of this, and because the passive warm-up might take too long, the *active warm-up* is needed. Active warm-up means the procedure of copying the current values of the state variables of the state oriented processes in the active unit to the corresponding state variables of the spare unit.

6.1 Requirements for active warm-up

The warm-up services provided by the virtual machine should be applicable to all or at least to most of the applications for cloning the dynamic state of the active computations i.e. for migrating the computations to the spare unit without stopping the active computation driven by the external events. This means that we need a stepwise algorithm that may take hold of the CPU for periods of no more than a few tens of milliseconds at a time, as we saw in Chapter 2. Between those periods, the applications are allowed to proceed to comply with the real time requirements.

The services should be as transparent to the applications as possible, except for invoking the services and the possibly imposed design rules on the applications.

The dynamic state data to be transferred from the active to the spare unit includes

- state variables in the data segments of heavy and lightweight processes,
- process control data,
- active time-outs,
- state data in the memory resident work files,
- additional memory buffer segments reserved by processes.

It should be possible to use the warm-up services also while the spare unit is in the SP-EX state so that a single computation can be returned to a consistent state if it has fallen out of it due to an error.

A DX 200 specific requirement was set in Chapter 2 indicating that the only source of the dynamic data for the warm-up services should be the active unit.

Naturally, the algorithm should cause as little disturbance to the active computations as possible, never cause an error in the active computation, and should stop when the spare unit has reached the consistent state.

The amount of data to be warmed up should be kept to a minimum and transferring data for which establishing isomorphism between the active and the spare units is practically impossible or too difficult should not be taken-up. We will call such data *unit specific*. For example, the stack segments contain absolute memory addresses and thus cannot be transferred, because memory allocation is done independently by each computer unit. Establishing a one to one mapping between active and spare variables is simple when the variables are allocated in the same relative addresses in the active and spare data segments. Additionally, for certain data types, the isomorphic mapping may be described in the service request to the warm-up services.

Only such state data has to be transferred by the warm-up algorithm, which cannot be just loaded to the spare unit at some point of its restart. Data files, which are part of the permanent steady state and also data units which carry state variables of processes that can be stopped for the duration of loading, can be handled by file loading. Examples of such processes are those whose sole purpose is to support operator commands.

The starting point at which the active warm-up would first be invoked was to be at some time during passive warm-up (see Section 5.3.3).

6.2 Atomic warm-up entity and warm-up order

The first problem to be tackled is to define the components that should be warmed up in an atomic action from the point of view of the applications. We will call these components the *warm-up entities.* This means that while the atomic entity is being warmed up, the contents of the entity may not change. Because of the complications caused by the memory resident files which are accessed without buffering for performance reasons, processes could not be taken as the atomic entities as such, even if all the software resources reserved by the process, like process control data, active time-outs, and allocated memory buffers were tied together with the process. The existence of such a memory resident file system means, that message passing is not the only means of communication between processes and that files can be used for this purpose, too. In our model of computations the memory resident files are represented by the set of data units of the system (see Notation 4.1).

It would be ideal, if the warm-up entity could be a process. The worst possible case is that all processes in a unit belong to the same warm-up entity. In that case, the warm-up would be impossible without stopping the computations for a long time.

The second problem is, whether it is necessary to warm up these atomic entities in a specified order and if so, on what account this order is defined.

The atomic warm-up entities should be defined in such a way that

1. The amount of data bound to the entity is kept as small as possible to avoid any real-time problems.
2. A warm-up order between two entities is defined only if it is necessary for the success of the algorithm.

To conclude let us formulate a definition.

Definition 6.1 A warm-up algorithm for a closed system of processes, e.g. a computer unit, is such that

1. In each step a spare warm-up entity is brought into a consistent state with the active one, and
2. A step of the algorithm is atomic from the point of view of the objects being warmed up, i.e. the contents of the warm-up entity may not change during its warm-up, and
3. After each step, if the algorithm is stopped, the warm-up entities already in a consistent state are kept there by the tools of the basic replication scheme taken into use before the algorithm was started, because the state of the entities already warmed up is not dependent on the state of the entities that will be handled after them, and
4. Applications may proceed between the steps of the algorithm, and
5. The algorithm ends when all the warm-up entities i.e. all processes and their data units in the spare computer unit have been warmed up once and are kept in a consistent state with the active unit by the tools of the basic replication scheme.

We will spend the rest of this chapter in showing that such an algorithm exists for our target environment.

6.2.1 Message passing relations between processes

Let us look at a message passing relation between *replicated state oriented processes* s_1, $s_2 \in \Sigma$, where Σ is a replication group. We will assume from the start that messages can be passed both ways between s_1 and s_2 because this is usually the case. Let s_1^{wo}, s_2^{wo}, and s_1^{sp}, s_2^{sp} be the corresponding processes in the active (i.e. WO) and spare units respectively. Figure 6.1 clarifies the situation.

In which order should s_1^{wo} and s_2^{wo} be copied to the spare unit to create the replicas s_1^{sp} and s_2^{sp} ? Clearly, if s_1 is warmed up first, s_2^{wo} could send a message *m* to s_1^{wo} , which could change the state of s_1^{wo}, before s_2 is warmed up. This would lead to an inconsistent state of s_1^{sp} because it did not receive message *m*. The same problem could clearly arise with process s_2 if the warm-up order were reversed.

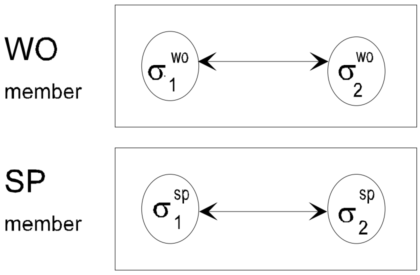


Figure 6.1 A Replicated message passing relation.

Before suggesting a solution to this problem we should note that binding the processes together is not an acceptable solution due to the requirement of keeping the warm-up entities small. This is, however, the only possibility if s_1 and s_2 exchange messages with unit specific data[17] both ways. That is why we have a

[17] see Section 6.1

design rule which is assumed to apply to our system:

DR1. The *design rule* should be followed: *Do not pass unit specific data in messages between processes which use warm-up services.*

The solution is expressed by a rule:

DR2. Every internal message from a non-warmed-up process to a warmed-up process in the active unit has to be replicated.

This is implemented by introducing a "*conditional replication attribute*" which is carried in the header of the internal messages of a replication group. This attribute makes sure that the message which cannot be sent by the spare process will be sent by the active one to the spare recipient.

Messages with the "conditional replication" attribute are replicated (the delivery code is changed to wo+sp) in the active unit under the following conditions:

a. The spare unit is in the SP state.
b. The replication state variable "Replicated_mode_ok"[18] of the sender process is false.
c. The "Replicated_mode_ok" of the receiving process is true.

The conditions mean that a process which is assumed to be in a consistent state (c) is dependent on the state of a process which is known to be in an inconsistent state with its active peer (b). The difficulty is overcome by delivering the necessary message from the active side.

Messages carrying this attribute are discarded when the sender is in the SP unit and the "Replicated_mode_ok" of the sender is false. When in both units the "Replicated_mode_ok" of the sender is true, this attribute does not affect the treatment of the message in any way, so inside a replication group member the logical computer address will remain "own -unit"[19].

This solution is an extension of the basic replication scheme and is always applicable when the design rule of *DR1* is followed.

If unit specific data is passed only in one direction, say from s_1 to s_2 , then clearly s_1 has to be warmed up before s_2 . Messages in the other direction could be replicated in the active unit according to rule *DR2*. Such designs are, however, not recommended because of the necessity to define a strict warm-up order between two processes.

6.2.2 Relations between a process and a data unit.

Let us look at a system of a process, its data units and the environment. First a definition is needed.

Definition 6.2 Let Σ be the system, *D* the data units, and *E* the environment, and $T = (N, \Lambda)$ the ACT for (Σ, *E*, *D*) and let $s \in \Sigma$, then

1. $\underline{\text{WRITESET}}$: $\Sigma \rightarrow \mathcal{P}(D)$ is a mapping from the set of Pids to the power set of all the data units of the process: $\underline{\text{WRITESET}}(s) = \bigcup_{n \in N} \underline{\text{writeset}}(s, s')$, where *n* denotes the

[18] This variable was introduced in Section 5.2.3

[19] This was introduced in Section 5.5.2.

nodes in the ACT for (Σ, E, D) and $s, s' \in S$ are the states of process s in all possible configurations of the ACT,

2. WRITESET(Σ) = $\bigcup_{\sigma\in\Sigma}$ WRITESET(s),
3. READSET: $\Sigma \to P(D)$ is a mapping from the set of Pids to the power set of all the data units read by the process, READSET(s) = $\bigcup_{n\in N} \bigcup_{a\in A}$ readset(current(s), a), where n denotes the nodes of the ACT for (s, E, D) and A is the set of messages,
4. READSET(Σ) = $\bigcup_{\sigma\in\Sigma}$ READSET(s),
5. $D_{\bar{\sigma}}$ = READSET(s) – WRITESET(s) is the set of data units of the environment of process s which may be read by the process,
6. By $s_1 \preccurlyeq s_2$ we denote that the left hand side has to be warmed up not later than the right hand side.

The set WRITESET(s) defined in 6.2.1 covers all possible data units the process may ever write to during its execution. A process needs address variables to access a data unit outside its own data segment. By this definition the set of data units of a process is tied to the behaviour of the process rather than to the static definition of the process. This approach reflects the indirect access method of the data units and the dynamic behaviour of a switching system. We believe that this helps us to minimise the size of the warm-up entities and opens possibilities for further optimisation on the level of implementation of the warm-up by passing additional information from the applications to the warm-up function.

The warm-up has to proceed in such a way that after a spare process has been warmed up it will be able to remain in a consistent state with the active peer while its state changes according to Definition 4.8. The basic replication scheme takes care of the states of the spare process when the additions of the wider Definition 5.8 are concerned.

One problem in defining the warm-up entities are the *read cycles* in a set of processes

$\Gamma = \{\pi_i \mid i = 0,1,..., n-1$ and $\forall\, i$: WRITESET$(\pi_i) \cap$ READSET$(\pi_{i\oplus 1}) \neq \emptyset \}$ where n is the length of the cycle and by $\oplus$ modulo-n sum is denoted.

Such a read cycle may also exist between sets of processes. Apart from the process being independent of a data unit there are three possible relationships between a process and a data unit: *read-only*, *read-write*, and *write-only*, and several processes can have access to the same data units. These sorts of problems lead us to a step by step approach in our further presentation. We will look at the data units of the application processes from the point of view of the warm-up algorithm i.e. we will avoid making any assumptions about the values of the data units or about in which transition the processes may write or read a particular data unit.

Lemma 6.3 Let there be a system of a process s, the set of data units D and the environment E. If the warm-up algorithm has no information about how the process uses its data units, then to meet the requirements of Definition 6.1, it is necessary for the warm-up of the process, its data units and the environment to satisfy the following conditions:

1. $\forall\, d \in D_{\bar{\sigma}}$ are warmed up not later than s,
2. WRITESET(s) is warmed up together with s in one entity.

Proof: If condition (1) of Lemma 6.3 is not followed, s could read a data unit in the environment just after warm-up. Then we must assume that s could fall into an inconsistent state because the data unit was in an inconsistent state.

If condition (2) is not satisfied and s is warmed up first and a data unit $d \in$ WRITESET(s) of the process after that, a consistency problem arises. Let us suppose that the warm-up of the data unit *d* has started and the data is in a message on its way to the spare unit but the data is not yet written on the spare side. Now s writes into *d* on both sides. Then this data in the data unit on the spare side will be overwritten by the older data in a warm-up message. The consistency error can start to propagate immediately when this data is read in the spare unit.

If a data unit $d \in$ WRITESET(s) is warmed up first and then s, a similar problem arises. Let us suppose that *d* has been warmed up, and s writes into *d* on the active side before s is warmed up. Then *d* on the spare side will not contain this data. Again the consistency error can start to propagate immediately when this data is read in the spare unit.

Consequently, both conditions have to be satisfied by the warm-up algorithm of a system which contains s and its data units.

The solution means that writes of process s into its data units during the warm-up of those data units are prevented from occurring and thus the consistency problem is avoided.

In Section 6.3 we will suggest an algorithm which allows the handling of warm-up entities containing a process and such data units.

6.2.3 Relations between two processes and a data unit

Let us consider the case when there is a process in the environment which writes into the data units of the system.

Lemma 6.4 Let there be a system of a process s and the set of data units *D* and the environment *E*. Let the warm-up algorithm have no information about how the process uses its data units, and let there be a process $\eta \in E$ in the environment such that WRITESET (s) $\cap$ WRITESET(η) $\neq \emptyset$. Then, if no more information is available about the behaviour of η , to meet the requirements of Definition 6.1, it is necessary that the warm-up of the process and the environment satisfy the following condition:

Processes s and η are warmed up together in one entity with the data units WRITESET(s) $\cup$ WRITESET(η).

Proof: Let us suppose that the condition is not satisfied, but instead s is warmed up first together with WRITESET(s) as required by Lemma 6.3 and η is warmed up only after that in a separate entity. What can happen?

It is not known whether process η may or may not proceed while s is being warmed up. Consequently, during the warm-up of s and its data units, due to item (3) of Definition 4.8 we must assume that an active process η may write into WRITESET(s) $\cap$ WRITESET(η) which is not empty. Such a write action may as well occur any time before η has been warmed up. Because the spare η is not in a consistent state, the same write action will not occur in the spare unit. Consequently, conditions (2) and (3) of Definition 6.1 are violated.

Clearly, if the warm-up order of s and η were reversed, the problem would remain the same.

Consequently, it is necessary to satisfy the condition of the lemma.

The condition of Lemma 6.4 means that the consistency problem is prevented from arising and that if the warm-up algorithm is unsuccessful for one of the processes in the warm-up entity, the warm-up of the whole entity fails.

Lemma 6.4 can clearly be generalised for the set of processes $\Gamma = \{\eta \in E \mid \underline{\text{WRITESET}}(\text{s}) \cap \underline{\text{WRITESET}}(\eta) \neq \emptyset\}$. All such processes $\eta \in \Gamma$ must be warmed up together with s and the data units of $\underline{\text{WRITESET}}(\text{s}) \cup \underline{\text{WRITESET}}(\Gamma)$. It is also necessary for the warm-up of the closed system $\{\text{s}, \eta\}$ of processes that condition (1) of Lemma 6.3 is satisfied for (s, E, D) and (η, E', D) i.e. it is satisfied for ($\{\text{s}, \eta\}$, E'', D).

Condition (1) of Lemma 6.3 can be generalised for a system of processes $\Gamma \subseteq \Sigma$, the set of data units D, and the environment E such that $\Gamma = \{\text{s} \mid \exists\, d \in \bigcap_{\sigma\in\Gamma} D_{\bar{\sigma}} \}$. Clearly the data unit that all the processes read has to be warmed up not later than any of the processes of Γ are warmed up.

Let us look at a system of processes Γ, the data units D, and the environment E such that $\Gamma = \{\text{s} \mid \exists\, d \in \bigcap_{\sigma\in\Gamma} \underline{\text{WRITESET}}(\text{s}) \cap \bigcap_{\eta\in\Gamma} \underset{\eta\in\Gamma'}{\bigcap,} D_{\bar{\eta}}$ where $\Gamma' \subseteq E\}$. Using the previous results, clearly, the system Γ of processes has to be warmed up in one entity not later than any of the processes $\eta \in \Gamma'$ are warmed up.

6.2.4 A simple read cycle

Lemma 6.5 Let there be a read cycle in a set of processes $\Pi = \{\pi_i \mid i = 0, 1, \ldots, n-1$ and $\forall\, i$ $\underline{\text{WRITESET}}(\pi_i) \cap \underline{\text{READSET}}(\pi_{i\oplus 1}) \neq \emptyset\}$ where n is the length of the cycle and symbol $\oplus$ denotes modulo-n sum. Then to meet the requirements of Definition 6.1 it is necessary that all the processes of Π are warmed up in one entity.

Proof: Condition (1) of Lemma 6.3 leads to the warm-up order: $\pi_i \preccurlyeq \pi_{i\oplus 1}$ which is a cycle. If the processes of Π are warmed up in separate warm-up entities starting from any π_i, according to Definition 6.1 it is required that after any step the warm-up may be stopped and objects which already were warmed up will stay in a consistent state. Let the warm-up be stopped after the first step of having warmed up π_i. Eventually process π_i may read data in $\underline{\text{WRITESET}}(\pi_{i\oplus n}) \cap \underline{\text{READSET}}(\pi_i)$ which has not yet been warmed-up and thus may be inconsistent. Then we must assume that π_i may fall into an inconsistent state. This is a violation of the above mentioned requirement of Definition 6.1. Consequently, the requirement of the lemma is necessary and the problem is avoided by warming up all the processes of Π in one entity.

6.2.5 Relations between a process and a set of data units.

The solution suggested in Lemma 6.3 item 2, can be used only as long as the total amount of data in the warm-up entity containing process s and all its data units $\underline{\text{WRITESET}}(\text{s})$ is not too large. For large sets of data units we need *another solution*. To make the solution easier we will set an additional restriction to the set of data units.

Definition 6.6 Let s be a process of an execution group Γ. When a data unit $d \in \underline{\text{WRITESET}}($ s) is not used to change the contents of any other data unit of the process, neither to change the state variables in the data segment of the process during a transition, nor is it used to generate the contents of any internal messages to other processes of Γ sent by the transition, the data unit d is *separate relative to the transition*.

If the data unit is separate relative to all transitions of the process, then it is *separate relative to the process*.

The definition implies that the data in a separate data unit $d \in \underline{\text{WRITESET}}(\mathrm{s})$ cannot be used by process s to change another data unit $d' \in \underline{\text{WRITESET}}(\mathrm{s})$ during any of the following transitions because there is no way to carry data from d to d' from transition to transition.

For the purpose of the modelling of warm-up procedures we may assume that the relationship between the process and a separate data unit is *write-only*.

Lemma 6.7 Let Δ_{S} be a set of all data units separate relative to process s. Let Δ_{S} be large so that it cannot be warmed-up together with process s and let $d \in \Delta_{\mathrm{S}}$. Then to meet the requirements of Definition 6.1, it is necessary for the warm-up of Δ_{S} to satisfy the following conditions:

1. s is warmed up in a warm-up entity Ω_0 and after that all the data units $d \in \Delta_{\mathrm{S}}$ are warmed up in the subsequent warm-up entities $\{\Omega_j \mid j = 1,...,m\}$ where Ω_j are disjoint subsets of Δ_{S} and

2. $\Delta_{\mathrm{S}} = \Omega_1 \cup \Omega_2 \cup ,..., \cup\ \Omega_m$

Proof: The state of the sets of data units Ω_j is affected by the write actions of process s. If condition (1) of the lemma were not satisfied and process s were not warmed up before a separate data unit $d \in \Delta_{\mathrm{S}}$ were warmed up, then according to Definition 6.1 the warm-up could be stopped after d were warmed up. Then process s on the active side could write into d whereas on the spare side this would not occur because process s were not warmed up yet. This is a violation of Definition 6.1. Due to condition (2) of the lemma all the separate data units will be handled by the warm-up algorithm.

This solution applies to a set of *separate data units* Δ_{S}. The restrictions of Definition 6.6 form a set of *necessary conditions* for the solution to be successful. The conditions seem very severe but such sets of separate data units are useful in a switching system. They can be used e.g. in a producer-consumer context for statistical and charging data collection. For example the charging counter file of a large number of subscribers is so large that it cannot be warmed up in a single entity but it can be split into a set of separate data units. Each data unit carries data for an independent object. Charging and other counters are state variables, statistical data collection for an object is a permanent state oriented computation. A single process carries out data collection for a set of objects, and computations for each object do not have anything to do with each other. Such data has to be transferred to a non-volatile memory periodically by a consumer process. Upon receiving a data message the statistic collection process validates the data and finds out which objects are concerned. For this purpose state data is not used and the collection process does not need state variables in its data segment for the collection computations. The actions on the objects are mostly simple, basically incrementing a counter or a packet of counters.

Note that the warm-up entity Ω_0 may also contain data units which are not separate. An algorithm which is in line with Lemma 6.7 has been implemented in the DX 200 system and its outline will be described in Section 6.4.

6.2.6 Generalisation

Definition 6.8 Let Σ be a closed system of processes, the data units D, and the environment E such that $s \in \Sigma$ and $\underline{\text{READSET}}(\Sigma) \bigcap \underline{\text{WRITESET}}(E) = \emptyset$. Let there be a partition $\Sigma = \bigcup_k \varphi_k$ where $k = 0,\ldots, p$ such that

(a) $\varphi_k = \{\forall s \in \Sigma \mid \exists\ s'$ such that $\underline{\text{WRITESET}}(s) \bigcap \underline{\text{WRITESET}}(s') \neq \emptyset\}$. Let φ_{k_0} be some φ_k and

(b) $\Gamma = \{\varphi_{k_0}, \varphi_{k_{h\oplus 1}} \mid \forall h\ 0 \leq h \leq m{-}1$ and $m \geq 2$: $\underline{\text{WRITESET}}(\varphi_{k_h}) \bigcap \underline{\text{READSET}}(\varphi_{k_{h\oplus 1}}) \neq \emptyset\}$

where m is the length of the read cycle and by $\oplus$ modulo-m sum is denoted. Then

1. If no information is available to the warm-up algorithm about how the processes $s \in \Sigma$ use their data units $\underline{\text{WRITESET}}(s)$, then the set of warm-up entities of Σ is:

   $W = \{\omega \mid \omega = \{\Gamma, \underline{\text{WRITESET}}(\Gamma)\}\}$, and

2. If there is a large warm-up entity ω' constructed according to (1) and there is a subset $\Delta \subseteq \omega'$ of data units of some of the processes $s \in \Gamma'$, $\Gamma' \subseteq \Gamma$ and it is known that the data units $d \in \Delta$ are separate relative to all such $s \in \Gamma'$ and not written to by any processes of $\Sigma - \Gamma'$ nor read by any processes of $\omega' - \Gamma'$, then there are warm-up entities:

   a. $\Omega_0 = \omega' - \Delta$ and

   b. Ω_j, $j = 1,\ldots,m$, where Ω_j are disjoint subsets of Δ and

   $\Delta = \Omega_1 \bigcup \Omega_2 \bigcup ,\ldots, \bigcup\ \Omega_m$.

Items (a) and (1) of the definition allow the construction of warm-up entities when there are no read cycles. Item (b) deals with the read cycles. Item (2b) defines when and how warm-up entities can be constructed out of separate data units. A break-down of Δ in item (2b) of the definition always exists because any set of data units can always be split into the constituent variables. Note that according to item (2) the data units of Δ can be only written to by processes of Γ' and possibly read by some processes $\sigma \notin \omega'$.

We need to apply our results to set Σ of all *state oriented processes*, $s \in \Sigma$, in a computer unit [Ar89]. This set is closed, so there are no processes in the environment which would write into the set of data units of Σ because stateless processes cannot change the contents of any data units by definition nor can they own state data which could be read by the processes of Σ. It is assumed that processes in other computer units have no physical means of writing into the data units of Σ because there is no shared memory. For the same reason the processes of Σ cannot read directly any data from other computer units. Consequently, the warm-up entities in a computer unit may be constructed according to Definition 6.8. We must still show that such an order of the warm-up entities exists that the warm-up on a computer unit level can succeed.

Notation 6.9 Let Σ be the system, D the data units, and E the environment, let $s \in \Sigma$, and let W be the set of all warm-up entities $\omega \in W$ of Σ, then

1. $\underline{\omega} = \{s \mid s \in \omega\}$ is the set of processes of ω

2. $\leftrightarrow = \{(s, s') \in \Sigma \times \Sigma \mid \exists\, \omega \in W: s \in \omega \text{ and } s' \in \omega\}$ denotes the relation of belonging to the same warm-up entity.

According to the theorem on equivalence relations and partitions, an equivalence relation on the set of all state oriented processes in a computer unit Σ determines a partition of Σ into disjoint subsets (i.e. $\underline{\omega}$) and a partition of Σ determines an equivalence relation (i.e. $\leftrightarrow$) on Σ. As an equivalence relation $\leftrightarrow$ is reflexive, i.e. a process belongs to the same warm-up entity with itself and symmetric, i.e. always $s \leftrightarrow s' \Rightarrow s' \leftrightarrow s$. As an equivalence relation $\leftrightarrow$ is transitive i.e. $s \leftrightarrow s'$ and $s' \leftrightarrow s'' \Rightarrow s \leftrightarrow s''$. This also follows from Definition 6.8 item (1) and items (3) and (5) of Definition 6.1 which mean that a process belongs only to one warm-up entity. Consequently, warm-up entities ω are equivalence classes in Σ.

Let us look at the order relation defined in the set of state oriented processes Σ:

> $\{(s, s') \in \Sigma \times \Sigma \mid$ such that s has to be warmed-up not later than s'$\}$, we have denoted this $s \preccurlyeq s'$ (Definition 6.2).

The equivalence relation for relation $\preccurlyeq$ is the relation of belonging to the same warm-up entity. Consequently, we can now formally define the relation $\preccurlyeq$ between warm-up entities.

Definition 6.10 Let W be the set of all warm-up entities $\omega \in W$ of the set of all the state oriented processes s in the computer unit, $s \in \Sigma$, constructed according to Definition 6.8 and let $\omega = \{\underline{\omega}, \underline{\text{WRITESET}(\omega)}\}$ where $\underline{\omega} \subseteq \Sigma$ or $\omega = \Omega_j$, $j = 0, 1, \ldots m$ where Ω_j are defined according to item 2 of Definition 6.8. The warm-up order is the relation:

$$\preccurlyeq = \{(\omega, \omega') \mid (\omega \cap \underline{\text{READSET}(\omega')} \neq \varnothing) \text{ or } (\underline{\text{WRITESET}(\omega)} \cap \omega' \neq \varnothing)\}^*.$$

This means that the warm-up order is the transitive closure of *read* relations between warm-up entities such that the latter reads data from the former and of *write-only* relations between warm-up entities such that the latter contains some separate data units of the former.

Note that for warm-up entities $\{\omega, \omega'\}$ constructed according to item (1) of Definition 6.8 $\underline{\text{WRITESET}}(\omega) \cap \omega'$ is always empty. A process s which has no data units, i.e. $\underline{\text{WRITESET}}(s) = \varnothing$ forms a warm-up entity of its own and can be warmed up in any order with respect to all other processes.

Lemma 6.11 Let W be the set of all warm-up entities $\omega \in W$ in a computer unit. Let $\preccurlyeq$ be the warm-up order defined in W according to Definition 6.10. The relation $\preccurlyeq$ is a partial order.

Proof: The relation $\preccurlyeq$ is reflexive because $\omega \preccurlyeq \omega$ and it is transitive by definition, i.e. $\omega \preccurlyeq \omega'$ and $\omega' \preccurlyeq \omega'' \Rightarrow \omega \preccurlyeq \omega''$. The relation $\preccurlyeq$ is also antisymmetric, i.e. $\omega \preccurlyeq \omega' \Rightarrow \neg\, \omega' \preccurlyeq \omega$, because

a. The write-only relation between the set of processes Ω_0 and their separate data units is antisymmetric: such data units can be written to only by the processes of Ω_0 and read by processes $\sigma \notin \Omega_0$. Clearly, if such $\sigma \preccurlyeq \Omega_0$ we have a read cycle and Definition 6.8, item (b) implies $\sigma \in \Omega_0$, which is a contradiction.

b. The read relation between warm-up entities is antisymmetric: Let us assume $\omega \preccurlyeq \omega'$ and $\omega' \preccurlyeq \omega$. Then by Definition 6.10 in this case:

i. $\omega \cap \underline{\text{READSET}}(\underline{\omega'}) \neq \emptyset$ which by Definition 6.2 implies $\underline{\text{WRITESET}}(\omega) \cap \underline{\text{READSET}}(\underline{\omega'}) \neq \emptyset$,

ii. $\omega' \cap \underline{\text{READSET}}(\underline{\omega}) \neq \emptyset$ which by Definition 6.2 implies $\underline{\text{WRITESET}}(\omega') \cap \underline{\text{READSET}}(\underline{\omega}) \neq \emptyset$,

By Definition 6.8 item (b) this is a read cycle between processes of ω and ω' which is a contradiction.

Consequently, $\preccurlyeq$is a partial order relation.

Let us assume that algorithms for the warm-up of a set of processes and their data units and the warm-up of a set of separate data units exist. What can we say about *sufficient conditions* for the warm-up of entities constructed according to Definition 6.8?

Lemma 6.12 Let Σ be the set of all *state oriented processes* $s \in \Sigma$ in a computer unit that has a spare and let W be the set of all warm-up entities, $\omega \in W$, in the unit constructed according to item (1) of Definition 6.8 and let $\underline{\omega}$ be the set of processes in ω. For the warm-up of any such ω to meet the requirements of Definition 6.1 it is sufficient that the following conditions are satisfied:

1. before the warm-up of ω all such warm-up entities were warmed up which are not later in the warm-up order than ω, and
2. during the warm-up of ω changes in the queues of processes of $\underline{\omega}$ are detected and the processes $s \in \underline{\omega}$ are not allowed to proceed, and
3. if changes occurred in the queues of processes of $\underline{\omega}$, the result of the warm-up of ω is discarded, and
4. new attempts are made until no changes in the queues occurred during the warm-up. Then the processes of $\underline{\omega}$ are allowed to proceed.

Proof: To show that the conditions of Lemma 6.12 are sufficient, we need to show that during the warm-up an initial isomorphic state can be created in the spare unit and that the system (ω, E, D) will remain in a consistent state in the spare unit indefinitely due to the basic replication scheme while state changes occur according to Definition 4.8.

The system (ω, E, D) is closed. I.e. $\forall\ s \in \{\Sigma - \underline{\omega}\}$: $\underline{\text{WRITESET}}(s) \cap \omega = \emptyset$. If this were not so then $\exists\ s' \in \{\Sigma - \underline{\omega}\}$ such that $\underline{\text{WRITESET}}(s') \cap \omega \neq \emptyset$. However, then according to item (1) of Definition 6.8 $\exists\ s'' \in \omega$ such that $\underline{\text{WRITESET}}(s') \cap \underline{\text{WRITESET}}(s'') \neq \emptyset$ and $s' \in \omega$ which is a contradiction.

Consequently, the processes of $\{\Sigma - \underline{\omega}\}$ can never change the configuration $C(n)$ of the ACT for $(\underline{\omega}, E, D)$. The only changes in the configuration that can take place are due to the actions of processes of $\underline{\omega}$. These changes are according to Definition 4.8 solely determined by the current state, the incoming messages and the read actions which determine the write actions of deterministic, well behaved processes.

Let $T = (N, \Lambda)$ be the ACT of the system $(\underline{\omega}, E, D)$. Conditions (2,3,4) of the lemma mean that an isomorphic copy of the global state of the system $(\underline{\omega}, E, D)$ expressed by the current configuration of the ACT: $C(n)$ can be produced in the spare unit using the data in the active unit. So, at the end of the warm-up the current states of the processes of $\underline{\omega}$ will be isomorphic in the active and the spare units.

Condition (1) of the lemma and the fact that, due to Definition 6.8, the warm-up entity ω is not a party in a read cycle with processes which are not included in ω, ensures that the simultaneous read actions of the active and the corresponding spare processes will produce the same data after the warm-up. Then it is up to the basic replication scheme extended with the conditional replication attribute[20] to deliver the same messages to the active and the corresponding spare processes of $\underline{\omega}$. Consequently, even if the warm-up of the spare unit were stopped after ω was warmed up, as required by item (3) of Definition 6.1, ω would stay in a consistent state in the active and the spare units. Consequently, the conditions of the lemma are sufficient.

Lemma 6.13 Sufficient conditions to meet the requirements of Definition 6.1 for the warm-up of entities Ω_i, $i = 1,\dots,m$ constructed according to item (2b) of Definition 6.8 are:

1. The warm-up order is not violated i.e.:
    a. Lemma 6.7 is satisfied for all Ω_i and the set $\underline{\Omega}_0$ of all processes, s $\in \underline{\Omega}_0$, i.e. $\forall\, i = 1,\dots,m$: $\Omega_0 \preccurlyeq \Omega_i$ and $\exists$ s $\in \underline{\Omega}_0$ such that $\underline{\text{WRITESET}}(s) \cap \Omega_i \neq \emptyset$, and
    b. Before the warm-up of Ω_0 all such warm-up entities were warmed up which are not later in the warm-up order than Ω_0, and
2. Write operations of s to Ω_i during the warm-up of Ω_i are detected by the warm-up algorithm, and
3. If a write action occurred during the warm-up of Ω_i, the result of the warm-up is discarded, and
4. The warm-up attempts of Ω_i are repeated until there is no write operation during the warm-up.

Proof: To show that the conditions of Lemma 6.13 are sufficient, we need to show that during the warm-up an initial isomorphic state can be created in the spare unit and that the system ($\underline{\Omega}_0$, E, D) will remain in a consistent state in the spare unit indefinitely due to the basic replication scheme while state changes occur according to Definition 4.8.

Conditions (2,3,4) ensure that an isomorphic copy of any Ω_i can be produced in the spare unit using the data from the active unit. If after the warm-up of Ω_i the warm-up algorithm is stopped, condition (1) of the lemma ensures then that the processes which write into Ω_i are already in a consistent state and will stay so according to Lemma 6.12. As the write actions of the processes of $\underline{\Omega}_0$ are the only cause of change in Ω_i the state of Ω_i will remain consistent and thus ($\underline{\Omega}_0$, E, D) will remain consistent. Consequently, the conditions of the lemma are sufficient.

Let Σ be the set of all *state oriented processes in a computer unit* that has a spare. For the warm-up of Σ it is sufficient to break it into warm-up entities according to Definition 6.8, warm up one entity in a step of the warm-up algorithm and do the steps in an order that does not violate the warm-up order.

[20] See Section 6.2.1.

The warm-up entities in a computer unit are defined by using write relations and read cycles between processes and data units which are portions of memory resident files. The warm-up order in the set of all warm-up entities in a computer unit is defined by using read and write-only relations between warm-up entities. The warm-up algorithm will end successfully if all the warm-up entities in the computer unit are successfully warmed up. The success of that algorithm will be conditional because of our basic replication scheme. The basic replication scheme will be responsible for keeping the processes which have already been warmed up in a consistent state. In Chapter 5 we saw that in this, the basic replication scheme is successful only under certain conditions or with a certain probability. In Chapter 5 we also saw that propagation of the replication errors of a closed system replicated according to the basic scheme is limited with respect to both dependent and parallel computations. For example the limited error cascading property of Lemma 5.10 applies to warm-up entities constructed according to item (1) of Definition 6.8 while the conditional replication attribute is taking care of the multicast deliveries of messages from the concerned warm-up entity to the processes of the other warm-up entities.

6.3 Active warm-up algorithm for a process

The algorithm handles the warm-up of entities of the type {s, WRITESET(s)} which is a special case of entities constructed according to item (1) of Definition 6.8. After the algorithm the state of the spare process and its data units {s, WRITESET(s)} is isomorphic to the state of the corresponding active process and its data units and the queues of the active and spare process are in a consistent state. According to Lemma 6.12 the state of the warm-up entity will remain consistent, if the warm-up order was not violated.

The algorithm is executed by a kernel program block, denoted here as W. A replicated process to be warmed up will be denoted as s, its active and spare instances as s^{wo} and s^{sp} respectively.

Starting conditions for the algorithm are:

- s^{wo} is in the main-receive point of execution,
- incoming message queue of s^{wo} is empty,
- saved messages queue of s^{wo} is empty,
- process s^{wo} has not prohibited warm-up.
- process s has not been warmed up yet, i.e. Replicated_mode_ok(s) = false.

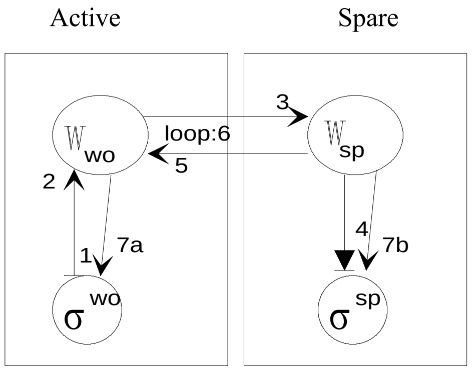


Figure 6.2. Warm-up of a process.

Figure 6.2 gives an outline of the algorithm. During this algorithm other processes except s in the active and spare computer units proceed normally.

The principal algorithm goes as follows:

1. Process s^{wo} is frozen. From this moment, if a message arrives in s ^{wo}, it will be put into the incoming

message queue and will not be processed.

2. State data of the warm-up entity in which s is a member is collected.
3. The data is sent to the spare unit. The first message also implies a freeze command of s^{sp}.
4. Process s^{sp} is created and frozen and data from the message is written for s^{sp}. The communication state of s^{sp} will be set to "reachable" and all messages coming to s^{sp} will be put into the incoming message queue from this point.
5. An acknowledgement is sent to the active unit.
6. If the incoming message queue of s^{wo} has remained empty, the freezing was successful. Otherwise the warm-up fails and the failure message is sent to the spare unit, where s^{sp} is returned to its initial state; s^{wo} is defrozen and can continue processing. A higher level function has to reattempt the warm-up of the process later.

   If the freezing was successful and not all of the data could be sent in the first message in step 3 of this algorithm, additional data is sent to the spare unit in consequent messages, the data is written in the spare unit and each message is acknowledged. Note that during this loop the incoming message queues of s^{wo} and s^{sp} do not need to remain empty.

7a. When there is nothing more to be sent, s^{wo} is defrozen and the status "Replicated_mode_ok" is set to true.

7b. When data from the last data message has been written, s^{sp} is defrozen and the status "Replicated_mode_ok" is set to true.

In a frozen state a process exists for the other processes and they can send messages to it. A frozen process will, however, not execute a single line of code and the kernel puts the incoming messages to the incoming message queue of the process. Those messages will be processed by the process when the frozen state ends.

During the steps (2) to (5) the incoming message queue of s^{wo} has to remain empty. The reason for this is that the copy of the same message could have been lost in the SP before s^{sp} is created. This way of handling the message queues works even if the events cannot be uniquely identified.

During the transfer of subsequent data items the messages to s will be buffered into the incoming message queues on both sides, which is a relaxation of condition (3) of Lemma 6.12. However, the incoming message queues of both of the active and the spare process are empty at the end of step (4) of the algorithm if step (6) does not fail. From step (4) onwards the queues will remain consistent if the basic replication scheme does not fail. Steps from 1 to 7 have to be executed with a high priority to minimise disturbance to the applications and to maximise the chances of success. However, the overall algorithm for all processes has to run in the background of the applications, so that the applications can proceed between the warm-ups of the warm-up entities. Consequently, the following rule for *priorities* must hold:

DR3	warm-up entity selection ≤ applications ≤ warm-up of an entity.

Is it always possible to traverse through all the warm-up entities? It is assumed that all the processes in a computer unit can always be found and the algorithm on the level of a computer unit loops through all the processes. The warm-up of data units has to be interleaved into this high level loop according to the warm-up order. This will be further discussed in Section 6.5.2.

6.4 Warm-up algorithm for a set of separate data units

Here we will describe, how the approach of Lemma 6.13 can be implemented in principle. Except for the condition that the data units have to be separate, there is one practical consideration that has to be taken into account. The warm-up will take rather a long time, and so due to real-time requirements the warm-up algorithm has to run in the background of the applications. In the DMX this is achieved by an appropriate allocation of priorities. Figure 6.3 gives an overview of the algorithm.

Legend:

- W_m – the kernel master process responsible for the file warm-up,
- W_m^{sp} – the SP instance of W_m,
- W_h^{wo}, W_h^{sp} – the WO and SP instances of the hand acting on file warm-up,
- $\Delta = \{d_i \mid i = 1,..., n\}$, the set of separate data units to be warmed up,
- s – the process to which the set of data units is related to.

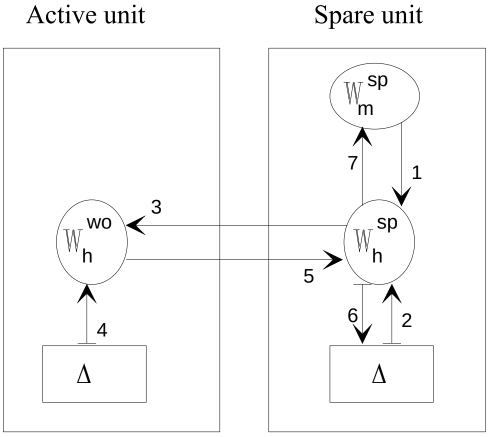


Figure 6.3. Warm-up of a set of separate data units.

Before the algorithm starts, process s has been warmed up. The computation skew Δr between s^{wo} and s^{sp} is assumed to be small. The algorithm goes as follows:

1. W_m^{sp} requests its hand to warm up a data unit d_i that can be carried in a message.
2. W_h^{sp} saves the data unit to be warmed up. This data is expected to be in an inconsistent state. We will denote the saved contents $v(d_i^{sp}) = u_{sav}$.
3. W_h^{sp} request the W_h^{wo} to send the data unit d_i.
4. W_h^{wo} reads the data unit.
5. W_h^{wo} sends the data unit d_i to W_h^{sp}.
6. W_h^{sp} reads the current value of the data unit in SP $v'(d_i^{sp}) = u_{cur}$, and

   if $u_{sav} = u_{cur}$, then

   The received data unit d_i is written into the set of separate data units on the SP side Δ^{sp}.

   Otherwise

   the warm-up of this data unit fails, and the warm-up has to be attempted again.
7. W_h^{sp} acknowledges the W_m^{sp}.

If this was not the last data unit, W_m^{sp} continues from step (1), otherwise the warm-up ends.

The test ($u_{sav} = u_{cur}$) in item (6) ensures that the write actions of s^{sp} to $d_i,^{sp}$ during steps (3), (4), (5) and (6) until the new contents is written, will be detected.

How large a computation skew Δr can be tolerated? If s^{wo} is later from s^{sp} than it takes to execute steps (2), (3) and (4) of this algorithm, then it may be that s^{sp} wrote something into d_i^{sp} just before step (1) that is not written into d_i^{wo} at step (4). It is assumed that skew Δr can be kept smaller than this.

Note that in steps (4) and (5) several messages could be passed if it does not take too long so that real-time problems would arise. This allows the data units to be bigger than the maximum message length.

To keep the condition that this algorithm runs in the background of the applications, to minimise the time of warm-up of a single data unit, and to ensure the safety period discussed in step (6), and thus to maximise the probability of success of the warm-up, priorities are set as follows:

DR4	priority(W_m) ≤ priority(Applications) < priority(W_h).

This rule ensures that s^{sp} can do nothing e.g. during steps (2) and (3). Before this algorithm can start, the hand processes W_h^{sp} and W_h^{wo} have to be allocated for the job. Protocols for this are trivial, and are not shown here.

This algorithm has been implemented in the DX 200 system.

6.5 Warm-up implementation issues

6.5.1 Atomic warm-up entities

In the process warm-up algorithm, in the DMX, the data to be copied to the spare unit comprises the following:

- Data in the process control block, like process kernel state variables, e.g. scheduling state and communication state,
- State variables in the statically allocated data segment of the process. Note that not the whole data segment needs to be warmed up because it may contain variables, the scope of which is limited to a transition.
- Dynamically allocated memory buffer segments. Such memory segments are always accessed with the help of a selector.
  - If the segment is in use, a buffer at least of the same size as in the active unit has to be allocated on the spare side, the selector of the allocated buffer is written to the corresponding variable of the spare process.
  - If the contents of the buffer is requested to be warmed up, it is, also, copied from the active unit to the spare side.
  - The value of the offset of the buffer pointer is copied from the active unit to the spare side.

- File handles[21]. If the file handle is in use in the active unit, then:
  - If the file was not opened at the spare process initialisation, the file is opened and the value of the file handle is written into the correct variable on the spare side.
  - If the file has to be warmed up, it is taken care of at this point.
- Time limits. In the DMX, the kernel System Master process provides replicated time-out services described in Section 5.5.1 The replicated active timers, as one of the software resources allocated to a process, are warmed up by the System Master process at the request of the warm-up control process. Time measurement counters in the data segment of an application process are considered to be like process state variables and in warm-up are treated as such.
- All the above data for possible other processes in the same warm-up entity.

Processes in the DMX can be frozen by setting their priority to a value, which is lower than the priority of the idle time process.

6.5.2 Warm-up order

The warm-up order is described in terms of *warm-up entity classes* in the current implementation. The membership of a process in a warm-up entity is given in a constant file. The file describes *warm-up entity classes* for which unique numbers are given and the membership is recorded for hand process types and for master processes which are considered to be of their own type. The warm-up control has to do the instantiation of the warm-up entity classes to find the objects to be warmed up.

In the current implementation, the instantiation of the warm-up entity class for a process family is done in the following order:

- the master process of the family is warmed up first,
- free hand processes in the last-in-first-out order, i.e. the process that was released last is warmed up first,
- reserved hand processes.

This implementation does not allow arbitrary combinations of processes to constitute the warm-up entities. This would require that warm-up entities and the warm-up order were described not in terms of classes but of individuals.

An issue related to the warm-up order is the nature of the relationships between a data unit and a process. The relation may be *static* or it may be *dynamically* established when the process is set to work. Static relation means, that the data unit is an extension of the data segment of the process. This may be the case e.g. when a set of data units are packaged into a file to save memory. In the warm-up algorithm, statically attached data units can be handled logically just as process data segments. These data units can easily be found by traversing all the processes assuming that each process will provide a descriptor of its data units. These descriptors get their values in process initialisation immediately after the process has been created.

A dynamically established relationship between a process and a data unit means that *the lifetime of the data unit is longer than the lifetime of the process*. This adds some complexities to the warm-up algorithm because some of the data units in a file may not be currently allocated to any process while the warm-up is taking place.

[21] This was introduced in Section 1.5.2.

Let us assume that we have a dynamically partitioned file $\boldsymbol{D} = \{ d_i \mid i = 1, ..., n\}$ and data units from this file are used occasionally only by processes of a single process family. In this special case a possible way of taking care of this is the following [Ar89]:

1. A dynamically partitioned file ***D*** is copied from the active to the spare unit. This takes care of the data units that are not allocated to any process during the warm-up. If, after this, one of those data units is allocated to a process, the process will read consistent data in the data unit in both the active and the spare units. If a process using a data unit d_i in file ***D*** is released during this phase, the allocated data unit d_i is warmed up during the process release operation. This makes sure that possible new non-allocated data units will be consistent after the process was released.

2. Free hand processes, possible users of ***D***, are warmed up in the first-in-first-out order, that is, in the order they will be allocated to a task. This is done by traversing the free hand FIFO of the involved process family. A process that is released during this phase is placed at the end of the FIFO, and the allocated data units of the process are warmed up during the release operation.

3. Now the allocated hand processes of the involved process family are traversed. With the process, the currently allocated data units are also warmed up. If a process is released during this phase, the allocated data units are warmed up, unless the process was warmed-up already i.e. "Replicated_mode_ok" was true.

This algorithm is rather complex and requires integration of warm-up and the basic replication scheme described in Chapter 5. Dynamically partitioned files can be warmed up using the approach of Lemma 6.13 if the data units are separate. Because of these two reasons, the above algorithm was not implemented in the kernel. The above order does not, however, contradict the warm-up entity class instantiation order adopted by the warm-up control for process families. Consequently, if it can be guaranteed that after hand release the same hand process is not allocated to a new task during warm-up, then dynamic allocation may be used, provided that the file is warmed-up before the process family.

6.5.3 Computer unit warm-up

Participants in the computer unit warm-up are the *local unit recovery control process*, the warm-up control program block, and applications that have *application specific warm-up functions*. To be able to call the application specific warm-up functions at any point in the sequence, a concept of *warm-up chain* is introduced. A warm-up chain is a sequence of warm-up entities in the warm-up order. The recovery requests the warm-up control process to warm up a warm-up chain. Knowledge about the warm-up order and about the warm-up entities is made available to the warm-up control with a constant data structure, which is placed in a memory resident file.

The warm-up control collects the detailed information about the warm-up entities from the *warm-up request messages* sent by the applications, usually at their initialisation phase.

To give an overview of the whole process, we will briefly look at the outlines of the unit warm-up sequence in a DX 200 signalling computer unit. Such sequences are described by the control data structures of the recovery control, which is a data driven program, and the sequences may be unit type dependent. The outlines are as follows:

1. The local recovery creates and starts the warm-up control process, and after that creates and starts applications that are the users of the warm-up services. The applications start sending warm-up request messages to the warm-up control process. The warm-up control

acknowledges each request and stores the information about the warm-up entities in a memory resident work file.

2. When the master processes are ready to allocate new hand processes, the recovery initiates the loose message synchronous mode, as described in Chapter 5.
3. Active warm-up of the first warm-up chain is started. This may be a request to the warm-up control process or a standard request to an application to execute an application specific procedure.
4. Upon request, the warm-up control acknowledges the recovery and starts warming up the processes and files of the warm-up chain concerned. The procedure is controlled by the master process of the warm-up control on the SP side. The hand processes of the warm-up control on the WO side provide appropriate state data to SP hands, who open the requested files, attach memory buffer segments to the process, and write the data into segments and data units of the process which is being warmed up etc. When all the warm-up entities have been successfully copied to the spare unit, the warm-up master process acknowledges the recovery control.
5. Item (4) is repeated for all warm-up chains.

Recovery control has time limits for the different phases of the unit restart and warm-up and an overall time limit. If these time-outs expire, it may again try certain phases of the warm-up procedure. If the overall time limit is reached, the unit is considered faulty and is taken down to the test state.

Because of the warm-up chain concept, if a process family is restarted by the recovery control in the spare unit, the whole warm-up chain in which the concerned process family is a member, has to be restarted and warmed up in the same recovery job. Then all the warm-up chains which come later in the warm-up order have to be restarted and warmed up.

6.6 Design rules imposed by warm-up services

Here we have collected some simple design rules and recommendations that should be followed in the applications that intend to use the currently implemented warm-up services. These rules give an impression about the applicability of the described services into an application.

- If you intend to use warm-up services, do not pass unit specific information in messages (*DR1*).
- Do not write unit specific information in files for which generic warm-up services are used.
- Hand processes of a normal process family can access the data segments of each other through the family LDT[22] (local descriptor table). In addition to this being a bad programming practice, it also leads to errors if the family uses warm-up services. Consequently, if *a process family has common data, it should be placed in a memory resident file* or if the hand processes only read common data, it can be located in the data segment of the master process.

[22] LDT is an Intel 80386 term. Each process family has its own LDT in DMX. The LDT contains descriptions of memory segments owned by the process family.

- If you have to use large memory resident files, restrict the access to the files with static partitioning or update the file only in such a way that the file can be split into separate data units.
- Do not use dynamic file partition allocation to processes.

  If it can be guaranteed that, after hand release, the same hand process is not allocated to a new task during warm-up, then dynamic allocation may be used.
- The data area in the data segment to be warmed up has to be continuous and the compiler has to be prevented from changing the order of variables by using a corresponding compiler feature. This helps to minimise the service degradation time when the exchange program package is changed.
- File handles and memory buffer pointers always have to be initialised to the NULL value when they are not in use.

Recommendations are another way to direct the thinking of the software designers. They are not compulsory, but will help the designer to avoid trouble. The following recommendations are in line with good programming practices and also follow from the theory of warm-up services.

- To avoid creating large warm-up entities, it is undesirable to transfer information between processes through files, especially in both directions. Instead, *message passing should be preferred*.
- Avoid read cycle type relationships between processes.
- Avoid designs in which two or more processes write state data into a file in a way that the file cannot be easily partitioned into a set of separate data units attached to each of the writing processes for warm-up.

7 Corrective Replication Tools

The basic replication tool guarantees a certain level of reliability in keeping the replicated computations in a consistent state, and the warm-up algorithms allow the computations to reach the consistent state in a reasonable time, when applied to telecommunication applications. There are, however, some problems left:

1. The basic tool cannot easily be applied to permanent computations.
2. What if the achieved level of reliability does not meet the requirements, or more strict requirements are set? What about applications which require instantaneous correctness?

Because of these problems, we will discuss additional tools that will at least increase the reliability of preserving the consistent state of the replicated computations and which can possibly also be applied to permanent computations. At best, even a tool that achieves conditional instantaneous correctness, could be added to the replication scheme. Only such tools will be considered, which will never decrease the possibility of the spare computation to remain in the consistent state. A highly desirable feature of the tools is application transparency. If not fully transparent, these tools should at the most be visible to the application code only through kernel procedure calls or asynchronous service invocations. For ease of implementation, it is also desirable to use as many of the existing generic mechanisms as possible. All the earlier identified real-time and performance requirements apply to these additional tools, also.

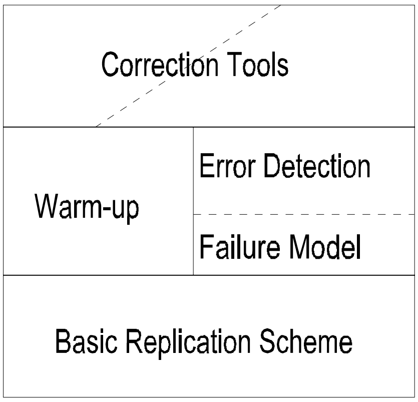


Figure 7.1 The Replication Machine.

All the tools that will be discussed, are based on the primary/standby scheme. This means that the optimistic failure assumption (Assumption 2.1 in Section 2.4) applies to these tools and so the whole intention of the tools is to try to keep the spare computation consistent with the active one. Figure 7.1 gives an overview of the structure of the whole replication machine we have in mind. The Basic Replication Scheme and the Warm-up have already been discussed. Some of the correction tools are based on error detection, so a model of replication errors is needed. Tools for error detection will invoke the error correction tools. Some of the Correction Tools may also be based on other principles than error detection. All the Corrections Tools may use the warm-up to carry out the corrective actions.

7.1 Coverage and Order of Correction

Before going to the actual tools we will discuss the applicability and conditions of use of any correction tools under our model of computations. By the *coverage of correction* we mean the set of processes and data units to which corrective actions are applied. We say that correction is *sufficient* if it covers all the processes and data units that may have been contaminated by the replication error. If we have no knowledge about the relationships between processes and data units, we do not know how the error may propagate in a computer unit. This is because of Lemma 5.10 and the errors discussed in Section 5.4.1 and the warm-up order. Then we must assume that a sufficient correction is to warm up the whole computer unit in case a replication error was detected. The question is: under what conditions is it sufficient to restrict corrective action to a subset of processes and data units in a computer unit. Because of what was earlier found out about the warm-up order, we must also ask in what order the corrective actions should be applied to a restricted error propagation area. Definition 6.8 leads us to assume that we do not have any knowledge about how a replication error may have propagated in a warm-up entity. That is why we will assume that an atomic corrective action covers, as a minimum, a single warm-up entity.

The results of Chapters 5 and 6 bring up the following lemma.

Lemma 7.1 Let W be the set of all warm-up entities $\omega \in W$ in a computer unit and Σ be a state oriented execution group involved in a computation in that computer unit and let the execution group be partitioned into replication groups Γ_j such that $\Sigma = \bigcup_{j=1}^{P} \Gamma_j$ and a replicated process $s \in \Gamma_j$. If there is a replication error in the spare member of a replication group Γ_j to meet the requirements of Definition 6.1 it is sufficient for the corrective actions to satisfy the following conditions:

1. The atomic corrective actions are applied to all the warm-up entities $\omega \in W$, where $\underline{W} \subseteq W$ are the processes in W, of the concerned computer unit such that the following rules starting from (a) are used repeatedly to add elements into W until no new entities can be found:
   a. ω is included in W if $\exists\, s \in \Gamma_j$ such that $s \in \omega$, and
   b. ω' is included in W if $\exists\, s' \in \omega' \cap \Gamma_i$ and $\exists\, s'' \in \Gamma_i \cap \underline{W}$, and
   c. ω' is included in W if $\underline{\text{WRITESET}(W)} \cap \omega' \neq \emptyset$, and
   d. ω' is included in W if $\exists\, \omega \in W$ such that $\omega \preccurlyeq \omega'$.
2. The conditions of Lemma 6.12 and 6.13 are met for the corrective actions on W. This includes that the order of corrective actions does not violate the warm-up order for W.

Proof: If the conditions are not sufficient then in the computer unit there can be an infected state oriented process γ such that $\gamma \notin W$ or an infected data unit d such that $d \notin W$. According to Definition 4.8 the future configurations $C : N \to (S_\gamma \times Q \times D_\gamma)$ depend on the current state of the process, incoming messages and read actions which determine the write actions of well behaved deterministic processes. This means that the ways a replication error can infect the system and its environment are: reception of an internal message sent by an already infected process, contamination of the reader by a read action from an already infected data unit, or contamination of the data units by write actions of an infected process.

Let us assume that before the replication error occurred in Γ_j the current state $C(n)(\gamma)$ of (γ, E, D) was consistent in the spare unit. Let us assume that the error has spread to γ by an internal message. However, due to (a), $\underline{W}$ contains all the processes of the initially infected replication group. Due to (b) it contains all the processes in all replication groups in which an already existing member of $\underline{W}$ is involved through another process. Consequently, $\underline{W}$ contains all such processes with which a process of $\underline{W}$ may communicate by internal messages and the spare $\underline{W}$ does not send messages to the environment because they are discarded by the kernel. So, the assumption that the replication error has infected γ by an internal message does not hold.

For the read-write, write-only and read relations the following statements hold:

1. Due to (d): $\forall\ \gamma\ \ \underline{\mathrm{READSET}(\gamma)} \cap W \neq \emptyset\ \Rightarrow \gamma \in \underline{W}$. Consequently there is no such process $\gamma \notin \underline{W}$ which could have been infected by the replication error through a read action from the contaminated data units.
2. Due to (c): $\forall\ d\ \ \underline{\mathrm{WRITESET}(W)} \cap d \neq \emptyset \Rightarrow d \in W$. So the infection could not have spread to $d \notin W$ through a write action of a process of $\underline{W}$.

Consequently, there is no such error propagation mechanism which could have infected a process $\gamma \notin W$ or a data unit $d \notin W$. Item (2) of the lemma ensures that the corrective actions are successful. So, the conditions of the lemma are sufficient.

Lemma 7.1 defines a method of how a sufficient area of correction can be constructed assuming that a single replication error has occurred. In practice we do not usually know where an error, if any, has occurred or we may want to provide for multiple replication errors and define a wider correction area by looking at the structure of the application. For example, we may want to make sure that all the spare processes involved in handling a single call with their data units or all the spare entities involved in executing a single operator command are in a consistent state at a certain time. Then we can use Lemma 7.1 to construct the sufficient area of correction by assuming that a replication error could have occurred in any of the processes of the largest state oriented execution group involved in the concerned computation and use Lemma 7.1 for each of those spare processes. The union of the resulting sets will be a sufficient area of correction providing for multiple replication errors in the application.

To conclude, a definition of conditional instantaneous correctness under the optimistic failure assumption[23] is useful.

[23] see Assumption 2.1.

Definition 7.2 Let $\Sigma \subseteq \Pi$ be a system of processes involved in a replicated computation and Π a system of processes of a distributed computing system such that processes in $\Pi - \Sigma$ are assumed to perform without replication errors which can not be traced back to Σ. Assumption 2.1 is assumed to hold for the distributed system. *Conditional instantaneous correctness of the replicated computation is achieved* in the distributed system in a state where it can be guaranteed that the state of any spare system of processes and data units (Π, Q, D) is consistent with the state of the corresponding active system.

7.2 Replication Error Detection

There are a number of errors that can be easily detected by the basic replication tool. These errors indicate an inconsistent state of the computation in the spare unit and are caused by incorrect message ordering and message loss errors, which are not tackled by the basic replication scheme. In Lemma 7.1 we discussed error states which are transparent to the kernel. To talk about error detection and correction, we need a high level *failure model* for errors in the computational state of the replicated computation. Figure 5.7 illustrates the state model of a replicated process and is a basis of the failure model.

The failure model basically incorporates two types of inconsistent states of a spare computation:

A. Error in the process kernel state i.e. the kernel state of a spare process is inconsistent with the kernel state of the active process. Most probably the spare process is in the suspended state, while the active process is involved in a computation. If the active process were free, this is also an error but does not influence any useful computations.

B. Inconsistency in the process application state. The state variables or the data units of the active and spare member processes have different values.

It is intuitively clear that error state B is often not stable and the more messages are processed in such a state by the replicated process, the more likely it is that error type B changes to error type A. This may happen e.g. because being in an incorrect state the process will not accept a message which was acceptable to the active process, and as there will not be an acceptable message, the process will try to release itself through a time-out procedure. However, there may be cases, when a spare process stays in an inconsistent application state for a long time.

An error of class A can easily be detected by the kernel primitives because the kernel state of the spare process member is changed only within the protocols of the basic tool. More specifically the following errors, during the execution of the algorithms of the basic tool, can be detected by the kernel primitives.

1. In the spare computer, a hand process quits but there is no release synchronisation message from the active unit during the time limit, and the hand is suspended. Most probably the computation in the spare unit has fallen into an inconsistent state.

2. The Pid search algorithm in the spare unit gives a different result than what is required by the message sent by the active unit. The locally found Pid is reserved and suspended. The corresponding computation may have fallen into an inconsistent state.

3. Messages with the wo+sp delivery option arrive at the spare unit or are sent by a process in the same spare unit, which are addressed to a free or suspended hand process. This may happen for several reasons, one of which is that the spare process is in an inconsistent state.

4. The hand to be allocated on the spare side is not free. The forced allocation corrects the error.
5. On the active side, the hand allocation acknowledgement does not arrive in time, or the current number of the allocation acknowledgement message is not correct.
6. The hand release acknowledgement message does not arrive in time, or the current number of the acknowledgement message is not correct, or the spare computer sends an acknowledgement with an error code.

7.3 Basic Error Correction Tool

A consistent state of the computations can be restored by warming up the computations. To make this possible, the kernel should indicate all detected errors to an analyser kernel process that should weight the errors and generate a *resynchronisation request* to the warm-up control program block.

Upon reception of the resynchronisation request (RESYN_REQ) the warm-up control executes the algorithm described in Appendix 1; ending with the RESYN_ACK, acknowledgement to the process that requested the resynchronisation.

In the algorithm of Appendix 1 retries are not shown to save space. Adding them to the algorithm is simple. The condition RESYNC ALLOWED in the algorithm of Appendix 1 is true when the hand has not prohibited warming up and it is in the main-receive point of execution. The object of resynchronisation is the warm-up entity to which the concerned process belongs. A warm-up entity in a general case may contain more than one process, but here we have shown only an algorithm for the case, when there is only one process in the warm-up entity.

In practice in the DX 200 context, resynchronisation is allowed only for warm-up entities such that the processes in that warm-up entity do not write into a set of separate data units outside of that warm-up entity. When the background large file warm-up would be required the unit warm-up is used instead. However, this is an implementation restriction, not a necessary condition of success as we have shown in Lemma 7.1.

This resynchronisation tool ensures that the *inconsistent state of the spare warm-up entity is corrected* under the optimistic failure assumption if

- there is no conflict between the algorithm and application processing, i.e. the queue of the active process remains empty during the necessary initial warm-up starting period,
- unrecoverable errors do not occur in the execution of the algorithm,
- the spare processes are found in the main-receive point, and the stacks of the spare processes are in a correct state.

The last conditions mean e.g. that the spare process may not be in an incorrect infinite loop, while the active member is working properly. Other software recovery and fault-tolerance techniques are assumed to handle such errors.

According to Lemma 7.1 correcting only the single warm-up entity in which the replication error initially occurred, is sufficient if

i. the processes of the infected warm-up entity do not send internal messages to other processes across the boundary of the infected warm-up entity, and

ii. there are no processes in the computer unit outside the concerned warm-up entity which could read a data unit in the infected warm-up entity, and

iii. the processes of the infected warm-up entity do not write into other warm-up entities.

The *basic error correction tool* will use the error information that is available to the kernel procedures (error type A). What can be gained with such a tool? One of its restrictions is that we cannot argue that the inconsistent state will *always* be detected nor can we definitely say how long it will take to detect the error. So, errors are detected with some probability. As a consequence, even conditional instantaneous correctness cannot be achieved by using only the error information that is available to the kernel procedures of the basic tool. In Chapter 8 we will see, what can be expected to be gained with error correction in terms of higher reliability. The advantage of this tool is that it is completely transparent to the applications. If this tool is applied to a computation for which the above conditions (i, ii, iii) do not hold, the success of the correction depends on whether an internal message (i), a read action (ii), or a write action (iii), actually occurred such that the error was spread outside the initial warm-up entity. Occurrence of such an event can be assumed to have a certain probability. It is not recommended that this tool is used while the above conditions (i, ii, iii) do not hold.

This tool has been implemented in the DX 200 system. The rest of the correction tools in this chapter are new possible enhancements to the replication scheme and have not been implemented in the DX 200 system.

7.4 Propagation of the corrective actions

In this section we will discuss a generic method which helps to propagate the corrective actions to the desired area of correction. Several correction tools may make the use of the same propagation method.

Correction may be launched by an external event or it may be induced by the system. In both cases correction is propagated through the correction area with the control flow as the *reaction*[24] is executed by the processes of the concerned execution group.

To implement this idea, a new message attribute called the *correction request attribute* is introduced. The idea is that when a message with this attribute is received, correction will be applied to the warm-up entity in which the receiver is a member. Propagation in a set of processes involved in a computation can be automated by *marking the processes* at the time of their creation. The correction request attribute is copied from the propagation message to the process and when a marked process sends a message during the transition started by the propagation message, the correction attribute is automatically sent together with that message. The correction attribute of the process is reset at the end of the transition. This mechanism ensures that correction propagation continues in the execution group until the boundary of the marking is reached, or a new external message is received. So, one process may be synchronised several times during the execution of a reaction. Resynchronisations after the first may be considered excessive. Their execution can be suppressed, e.g. by setting a short *time-out* for the process at the same time as the correction attribute of the process is reset. Before the time-out expires, additional corrective actions for the concerned process would not be allowed.

[24] see Definition 4.13.

7.5 An asynchronous error correction tool

In trying to improve on the performance of the basic error correction tool, the next obvious step is to try to detect errors in the application processing (error type B), initiate corrective actions and propagate the correction through the computation and thus achieve even higher reliability. This can be done e.g. as follows:

1. A kernel procedure call `check_replication_error`*(...)* is provided for applications. The procedure checks whether correction is allowed and if the process belongs to a warm-up entity, which has other process members; those processes are frozen. Checksum is calculated over the concerned warm-up entity. The checksum is sent to the error analyser kernel process in a message. The other processes in the warm-up entity are defrozen.
2. In the spare unit, the other processes of the concerned warm-up entity are frozen and the checksum is calculated over the warm-up entity. The checksum is sent to the error analyser kernel process and the processes are defrozen.
3. Correction may be propagated through the correction area using the method described in Section 7.4. So, the `check_replication_error`*(...)* procedure is used upon reception of a message carrying the correction request attribute and this attribute is sent to the next processes in the reaction.
4. When the error analyser gets a checksum message, it sets the time-out T_{cs}. If another message concerning the same warm-up entity arrives within the time-out, checksums are compared. Inequality for a warm-up entity with one process member is an error. For a multiple process warm-up entity, inequality means that an error is suspected. Also, if the time-out T_{cs} expires, an error is suspected.
5. Upon deciding on an error indication or an error suspected indication, the analyser process generates the RESYN_REQ for the warm-up entity concerned. This request message is handled as described in Appendix 1.

Could the tool be used as well at the beginning of a transition as at the end of a transition? When used at the end of the transition, if the spare process was not in a consistent state at the beginning of the transition, it may not be able to send the same (internal) messages as the active process. When used at the beginning of the transition, the situation is basically the same because corrective actions are not guaranteed to take place before the execution of the transition. This means that the error may cascade. However, we know that error cascading is restricted to the limits defined by Lemma 7.1. The error state of all the processes involved in a reaction is detected, if the tool is applied to all the processes during the execution of the reaction, so the tool will generate resynchronisation requests for all the processes in an error state. According to Lemma 7.1 the correction order, which may be different from the order of involvement of processes in the reaction, has to be such that it does not violate the warm-up order.

Of all tools suggested until now, this is the first tool that targets keeping the states of permanent computations consistent. This tool is able to detect all the errors in our failure model, stated earlier. However, if checksums are calculated by the kernel library routine, we have a modularity problem, because the library until now does not have any knowledge of the warm-up entities. To keep the protocol as efficient as possible, we could sacrifice coverage by restricting the checksum calculation to the vital state variables only. Another possibility is to make the warm-up entity descriptions part of the static structure of the process, using language tools and thus make warm-up entity descriptions available to the kernel.

7.6 Time-out triggered synchronisation

The more complicated and the longer a computation is, the more unreliable is the basic replication tool as we will show in Chapter 8. To ensure that all the hardware and software resources in the system are always released after use, all resources may be required to be allocated only for a certain period of time, with the necessity to renew the service provider allocation request, if the resource is still needed at the end of that period.

This brings up the idea that when reservations of software resources like hand processes are refreshed they could be resynchronised at the same time. When resynchronising a process to ensure the result, all the other processes involved in the same computation should be resynchronised at the same time. For this two things are required:

1. A kernel procedure `refresh`*(hand_Pid, additional_period)* is provided for the master processes. This procedure refreshes the allocation of the hand process for an additional period and resynchronises the process in the spare unit.
2. The refreshing is *cascaded* through the whole replicated computation.

The `refresh`() kernel procedure works as follows:

WO unit

Master process M_{wo} receives renew allocation request message and calls `refresh`*(R, dT)*

```
refresh(R, dT)
    Update allocation time-out by dT
    send RESYN_REQ to Warm-up control
    receive with time-out RESYN_ACK
    if RESYN was SUCCESS
        cascade time-out of R
    fi
    return
END refresh
```

SP unit

M_{sp} receives renew allocation request and calls `refresh`*(R, dT)*

```
refresh(R, dT)
    Update allocation time-out by dT
    receive with time-out RESYN_ACK
    if RESYN was SUCCESS
        cascade time-out of R
    fi
    return
END refresh
```

Upon reception of the RESYN_REQ the Warm-up control executes the algorithm described in Appendix 1. Note that the RESYN_REQ messages are assumed to be served sequentially by the Warm-up control.

The master process renews the allocation of the service provider hand for a computation on request from the original service user process. The user process will have an active time-out, at the expiration of which it will know that a *renew allocation request* message has to be sent to the master process of the service provider. Cascading to the following service provider in the allocation order can be triggered by setting the corresponding time-outs of the refreshed process to zero. After the renewal of the allocation of the service provider hand, the hand will receive the time-out message and send the renew allocation request message to the following service provider in the allocation order. In order to make the resynchronisation cascade through the whole set of processes involved in the computation, cascading has to start from the topmost user process in the allocation order in each of the involved computer units.

The hand allocation time-outs form a natural marking for the set of processes to be covered by correction. Setting the time-outs to zero acts as a replacement for the correction request attribute that was introduced in Section 7.4.

For this correction tool the correction order is the same as the execution order of the `refresh`() procedures because the resynchronisation requests are served in the first-come-first-served order and it should not contradict the warm-up order in any of the computers involved. The cascading order which is the same as the initial service provider allocation order is guaranteed to be identical to the execution order of the `refresh`*()* procedures for the processes only as long as each service user has only one service provider. Consequently, in this simple case the *cascading order should not contradict the warm-up order in an involved computer unit*. If a service user has several service providers, the order of correction of those service providers is dependent on process priorities. Then if it is required that none of the possible correction orders of the service providers contradicts the warm-up order, the correction may be successful.

To exclude the possibility of the cascading starting from any other process than the topmost user, it is clearly sufficient that the following condition applies to the renewal time limits:

$$\forall\ i, j, k\ \{T_{ij} + \Delta t^{e},_{i}\ <\ T_{jk\cdot}\}$$

where T_{ij} is the time limit of the service user process i for the renewal of the allocation of its provider process j in the allocation order,

Δt_{i}^{e} is the execution time for resynchronisation of process i and cascading the resynchronisation to process j ,

T_{jk} is the time limit for the renewal of the allocation of a service provider k of process j in the allocation order.

The renewal time-outs may not be in the same phase but it is not necessary to set additional conditions for the phases. If cascading starts from any other process than the topmost user, there is a high probability that only part of the possible inconsistency problem will be corrected. This means that computation resources are wasted and thus it should not be allowed to occur.

It is assumed that the `refresh`() procedure is called relatively infrequently. To ensure that the renewal of allocations will not waste computation resources unnecessarily, e.g. in call processing, the renewal time limits of about two times the average length of a call have to be adopted. This means that although with this tool all replicated computations, irrespective of their length, are going to remain synchronised approximately equally reliably, we are still far away from instantaneous correctness. The reliability behaviour of this tool will be discussed more thoroughly in Chapter 8.

Except for checksum optimisation described in the asynchronous error correction tool, an additional optimisation could be suggested for this algorithm. Namely, if a process has not received any messages after the latest resynchronisation, a new resynchronisation operation is clearly not required. The fact that the process has not received messages after the latest resynchronisation is easy to verify by setting a flag when a message is received and resetting the flag at each resynchronisation.

7.7 Safe Transitions

In the requirements analysis we saw that high reliability in call processing is required for established calls, while the bulk of processing takes place during the call set-up phase. In transaction processing, only at commit time is instantaneous correctness required; only the committed states have to be consistent. In our computation model, this corresponds to the idea that consistency of a replicated computation while processing certain events is more important than while processing other events. Examples of such important events in call processing are *alerting*, *connect* and *disconnect*. Alerting means that the called subscriber is being alerted, connect that the through connection has to be established and call charging started, and disconnect that the charging has to stop and all resources should be released. Note that in the DX 200 these are only three messages out of about two hundred that have to be processed in connection with a call.

To be able to focus our efforts on preserving consistency of a replicated computation while processing selected events, we will introduce the concept of *safe transition*.

Definition 7.3 Transition of a replicated process is *safe* if the final state of the transition is consistent and both the active and the spare process instances are guaranteed to send the same messages in the same order during the transition.

Obviously this property can be extended to a set of processes involved in a reaction using the method described in Section 7.4.

To guarantee state consistency of a transition, resynchronisation of the process members is required. Two possible implementations of the concept of safe transition can be seen. We will denote these implementations by A and B and discuss their properties.

A. Resynchronisation takes place at the beginning of the transition in the `Begin_Safe_Trans`() kernel procedure which is called from the `main_receive` procedure. The transition is executed as required by the basic replication scheme. Messages sent during the transition may carry the correction request attribute as discussed in Section 7.4.

B. Resynchronisation takes place only at the end of the transition. The `Begin_Safe_Trans`() kernel procedure sets the correction attribute of the process. All the messages sent during the safe transition are buffered in the outgoing message queue by the `send` kernel primitive. These messages may carry the correction request attribute. The outgoing message queues of the process members are synchronised with all the other data in the concerned warm-up entity by the `End_Safe_Trans`() kernel primitive. This primitive also resets the correction attribute of the process and sends the messages in the outgoing message queue.

We will first describe in more detail the approach A. There are several implementation possibilities, but in Appendix 2 we have shown one possible implementation of the procedure `Begin_Safe_Trans`().

In Appendix 2, we have not shown the reattempts of message sending between the two computer units, nor have we shown the automated propagation of correction. To add the automated propagation the `Begin_Safe_Trans`(), instead of cascading time-outs, would have to set the correction attribute of the process, the messages sent by the transition would then carry the correction request attribute, and the `End_Safe_Trans`() would have to reset the correction attribute of the process. Adding reattempts would effectively copy Figure 4.2. The service provided by the synchronisation server in this case differs slightly from the service invoked by the RESYN_REQ message to the warm-up control. Here the incoming message queues are synchronised to contain at least one message, namely, the message that started the safe transition and carried the correction request attribute. Another difference is that we have shown here the checksum optimisation of the algorithm. To minimise the number of messages in the algorithm in the absence of errors, the spare unit sends the SAFESYN_ACK message to the process which is being resynchronised and is executing the kernel library primitive. In this most common case with no errors, the algorithm requires two messages between the computer units (SAFESYN_SP and SAFESYN_ACK) and one internal message (SAFESYN_REQ) to be passed upon reception of a correction propagation message. The fourth message is the time-out message in the active unit but it is passed outside the synchronisation point. If the whole algorithm were executed in the name of the process being resynchronised, i.e. directly in the kernel library code, only the internal messages could be squeezed out and still the two messages between the computer units would be required.

In Appendix 2 we have also shown the possibility of adding time-out cascading in the `Begin_Safe_Trans`() primitive. The idea is that time-out cascading could be used, if it is not possible to use the propagation of correction with the reaction. Naturally an application should always be addressed only by one of these methods, or resynchronisation could happen twice successively.

The condition SYNC-ALLOWED in the algorithm of Appendix 2 assumes that the calling process R_{wo} is at the safe synchronisation point and that safe transitions have been allowed by the recovery control. The algorithm is shown only for the case, where there is only one process in the warm-up entity. The algorithm for the case, when there are several processes in the warm-up entity, is similar. Synchronisation of those processes is done with the assumption of empty incoming message queues.

We can see that the algorithm for a *deterministic transition* of a process under the optimistic failure assumption succeeds if

i. there are no unrecoverable errors during the execution of the algorithm of Appendix 2,

ii. the spare process is in the main-receive point or waiting for synchronisation, and the stack of the process is in a correct state. This allows e.g. for the process to be in the suspended state.

These conditions amend the conditions under which the synchronisation invoked by the RESYN_REQ succeeds by allowing for the spare process to also be in the waiting for synchronisation state, and by the implementation of the check whether the queue of the active process has not changed during the initial critical period.

In approach B, the `Begin_Safe_Trans`() is simple. The `End_Safe_Trans`() primitive is more interesting. It goes as follows:

WO unit

```
End_Safe_Trans()
    send RESYN_REQ to warm-up control
    receive-with-time-out RESYN_ACK
    if RESYN was OK
        cascade time-outs
    fi
    send messages in outgoing queue
    reset correction attribute of $R_{wo}$
    return
END
```

SP unit

```
End_Safe_Trans()
    receive-with time-out RESYN_ACK
if RESYN was OK
        cascade time-outs
    fi
    send messages in outgoing queue
    reset correction attribute of $R_{wo}$
    return
END
```

The warm-up control executes the same service as described earlier upon reception of the RESYN_REQ message, except that additionally the outgoing message queue is transferred from the WO to the SP unit. Here we have also shown the possibility of adding time-out cascading in the `End_Safe_Trans`() primitive. The idea is that time-out cascading could be used if it is not possible to use the propagation of correction with the reaction. Naturally, as in approach A, a process should always be addressed only by one of these methods, or resynchronisation could happen twice successively.

Which one of the approaches, A or B, should we prefer? Approach A is more modular because all that is needed are two new kernel primitives and the correction request attribute. Approach B requires additionally a new queue to be attached to every process and the buffering of messages into that queue to be added to the `send` primitive. Approach A does not have the side effect present in approach B. Namely, approach B changes the order in which the process executing a transition accesses files and sends messages. In a safe transition all file access operations are executed normally but the messages are not sent until after all those access operations have taken place. If the same transition were executed without the safety property, the sending of messages and file operations could occur in an arbitrary order. This side effect is undesirable, although if good programming practices are followed, applications should be insensitive to it. As a consequence, *approach A is preferred* for implementation, and in sequel we will address its properties.

7.7.1 Properties of the safe transitions tool

Consistency of the whole computation is not automatically guaranteed after the safety propagation. To address this problem, we need a definition.

Definition 7.4 Computation is safe if all its transitions are safe and the execution order of the corresponding active and spare transitions is identical.

The order of involvement of processes in a *sequential reaction* is the same as the order of correction, i.e. the order in which the synchronisation requests are served by the synchronisation server assuming that the requests are handled according to the first-come-first served principle. If in a safe transition the correction request is propagated to a number of processes, the order of correction of those processes is dependent on priorities. We will ignore the priorities and assume that any of those orders are possible.

Under the optimistic failure assumption *conditional instantaneous correctness of the replicated computation in the boundaries of the replication group is achieved* after the execution of the safe transitions by all the process pairs of the replication group in a complete reaction i.e. in the absence of new external events, if

C0 the replication group is closed and does not read data units of processes which do not belong to the replication group

C1 correction request attribute is propagated through the largest state oriented execution group of the replication group, and

C2 resynchronisation succeeds for all the processes in that group, and

C3 none of the possible correction orders contradicts the warm-up order. This means that if process σ' may follow process σ in the correction propagation order, processes σ and σ' may either be unrelated in the warm-up order or $\sigma \preccurlyeq \sigma'$, and

C4 the other requirements of Lemma 6.12 and 6.13 are met by the corrective actions.

In other words, a reaction in a closed replication group that does not read the data units of processes outside the replication group, is safe if all the transitions in the reaction are safe and the transition execution order in the reaction does not contradict the warm-up order.

In our analysis we did not use the knowledge about how the correction propagation messages are sent; i.e. what destination delivery code is used. This is why we may assert even more: A reaction in a closed state oriented execution group that does not read the data units of processes outside the execution group, is safe if all the transitions in the reaction are safe and the transition execution order in the reaction does not contradict the warm-up order in any of the involved computer units.

Using Lemma 7.1 we may conclude that a reaction achieves conditional instantaneous correctness in a system if

(i) the reaction initiated by a message with the correction request attribute involves all the processes which belong to the possible replication error propagation area of the computation which is formed as a union of all the sufficient correction areas of all the processes in the largest state oriented execution group involved in the computation, and

(ii) correction succeeds for all warm-up entities in which at least a process involved in the reaction is a member,

(iii) the order of correction does not violate the warm-up order.

Note that the above two results as such are independent of the optimistic failure assumption. It only happens to be that implementation in our target environment is possible only under the optimistic failure assumption.

During the warm-up, we had to replicate messages in the active unit with the help of the "conditional replication" attribute when the sender had not yet been warmed up. We avoid the problem here because the synchronisation takes place at the beginning of the transition.

The conditional instantaneous correctness property claimed above for a replication group, an execution group and a computation in a system is such that each process in the propagation order knows about the success only in the boundaries of its warm-up entity. It can, however, pass forward the success information in a propagation message. *Global consistency in the absence of new external events is implicitly known to the last process in the propagation order of a sequential reaction.* If the computation has parallel branches ending with a number of downmost service providers, knowledge about global consistency is dispersed.

This safe transition tool could be used in a transaction processing subsystem in the switching environment provided that the subsystem implementation avoids parallel branching at commit time. However, this method had not been implemented, when the implementation of the transaction processing subsystem of the DX 200 began, and that is why an explicit message exchange between the active and spare members of a process in the transaction commit protocol was used.

7.8 Comparison of the correction algorithms

Now, we have an abundance of tools. Do we need them all or could we take just one of them? To answer this question we have to make clear the properties of the tools and decide whether the tools compete with or supplement each other. Here we will compare the tools by their functional, ease of use, and performance cost overhead properties. In Chapter 8, the probabilistic properties of the tools will be addressed. The tools to be compared are:

A. The basic error correction tool
B The asynchronous error correction tool
C. The time-out triggered synchronisation
D. The safe transitions tool

Functional properties:

A. Instantaneous correctness is not achieved nor sought. The application cannot influence the tool because it is fully transparent to the applications. Processes involved in a computation are addressed by this tool independently. Correction efforts can be focused on errors or suspected errors.

B. Instantaneous correctness is close but not achieved nor sought. Correction efforts have to be focused on important events. The warm-up order relations between processes restrict the possibilities of using this tool in a similar way as is the case with the safe transition tool. The synchronisation error indications are put into a queue by the error analyser in the transition execution order, the error analyser sends the resynchronisation requests to the warm-up control in the same order, and the requests are acted upon in this order. This means that similar conditions of success as (i, ii, iii of Section 7.3) apply in this case also. The difference is that there should not be any new external events before all the resynchronisations have been executed.

C. If the resynchronisation succeeds for all the warm-up entities of all the processes in the cascading order, the cascading order does not contradict the warm-up order, and during cascading there were no new external events, the state of the computation restricted to the replication group is consistent after all the resynchronisation operations have taken place. However, the applications cannot focus this correctness on the important states of the computation.

D. Local and implicit global correctness of sequential computations is achieved. The safety propagation can cross the replication group and computer unit boundaries if it is desirable for the application. Correction efforts can be focused on carefully selected events but not on errors only.

Ease of use:

A. Easiest to use. Does not require any additional design or programming efforts from the application designers. However, the tool is not likely to help with permanent processes.

B. Can be used for periodical and for permanent processes. Similar to the safe transition tool.

C. Requires certain design rules to be followed in the applications. All the processes should be reserved for finite intervals. The process reallocation time limits should be set accordingly; the process allocation order and the warm-up order should not contradict each other. The tool is visible to the master process code.

D. The tool can be made easy to use for standard applications, like call processing; and replication can be addressed as a programming-in-the-large issue as we will show in Chapter 9. The correction propagation order should not contradict the warm-up order and the high level designer has to select the important events to which the tool is applied.

Performance cost overhead:

Applied to call processing, the tools use CPU capacity as follows:

$$A < C < B < D.$$

A. CPU capacity is used only when errors occur or when they are suspected to have occurred in the basic replication tool. Clearly this happens very infrequently and thus the performance overhead is negligible.

C. Requires at least one checksum optimised resynchronisation operation for all state oriented processes involved in processing long calls.

B. Requires at least one checksum calculation per process instance, one external message, and one internal message. Should be applied to all calls and to all state oriented call computation processes in duplicated computer units.

D. Requires at least one checksum optimised resynchronisation operation for all currently replicated processes involved in processing any calls. If the process members are found to be consistent, this adds up to counting the checksums over the concerned warm-up entity in both units and passing one internal message in both units and two external messages between the active and spare computers. Qualitatively, if we assume five state oriented processes, of which three are 2*N* redundant, to be involved in processing a call, one safe transition in each of those processes would add 12 messages to the about 200 call processing application messages. This is about a 6% increase but the actual performance overhead still heavily depends on the relative complexity of the application transitions and the resynchronisation transitions.

Conclusion:

Of the suggested tools C competes with A and B, while B complements A. However, there seem to be differences in the applicability of C compared with A and B. That is why we leave the final decision about A, B and C to the reliability analysis. The safe transition tool aims at conditional instantaneous correctness and thus cannot be replaced by any of the other tools.

8 Reliability Modelling and Analysis

To evaluate the basic and different corrective replication tools, we will use Markov´s reliability modelling techniques and introduce Markov´s reliability models for a call computation under different conditions. This modelling technique for discrete-event, continuous-time models with constant hazard functions is well known and has been represented e.g. in [Ne87].

8.1 Basic Reliability Model

Figure 8.1 shows the model of a call computation, when only the basic replication tool is used. A call is chosen as the modelling object because availability performance requirements exist for call handling as presented in Chapter 2.

Notation 8.1 State probabilities: P_0 - probability that a call that is in progress or has been established, is consistent in the spare unit, P_1 - probability that the call has been successfully disconnected in the active and spare computers, P_2 - probability, that an inconsistency error has occurred in the spare computation.

Transition intensities: α - error occurrence intensity of the spare computation, μ -call disconnection intensity.

In the model it is assumed that if an error in the state of the spare computation has occurred, it was at the beginning of the computation and that call disconnection always also clears the state of the spare computation. Constant distributions for the intensities are used for simplicity.

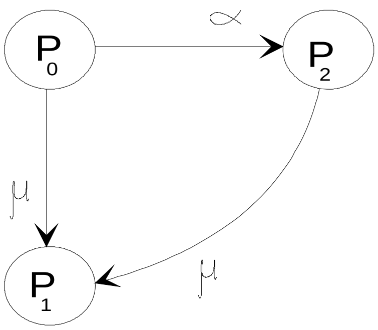


Figure 8.1 Basic Reliability Model for a call computation.

With this model the state probabilities at time $t + \Delta t$ can be expressed using the probabilities at time t and state transition probabilities by the following matrix equation:

$$[P_0(t+\Delta t), P_1(t+\Delta t), P_2(t+\Delta t)] = [P_0(t), P_1(t), P_2(t)] \text{ X } \boldsymbol{P} \qquad (6)$$

where ***P*** is the transition probability matrix:

$$\boldsymbol{P} = \begin{bmatrix} 1\text{-}(\alpha + \mu)\Delta t & \mu\Delta t & \alpha\Delta t \\ 0 & 1 & 0 \\ 0 & \mu\Delta t & 1\text{-}\mu\Delta t \end{bmatrix} \qquad (7)$$

A set of simultaneous differential equations follows by taking the limit as Δt approaches 0:

$$[P_0'(t), P_1'(t), P_2'(t)] = [P_0(t), P_1(t), P_2(t)] \text{ X } \begin{bmatrix} \text{-}(\alpha + \mu) & \mu & \alpha \\ 0 & 0 & 0 \\ 0 & \mu & \text{-}\mu \end{bmatrix} \qquad (8)$$

We are interested in finding the solution to P_2 because it represents the unavailability performance of the basic replication model. Taking that initially $P_0(0) = 1$, $P_1(0) = 0$, $P_2(0) = 0$ and solving the equations produces:

$$P_2 = e^{-\mu t} - e^{-(\alpha+\mu)t} \qquad (9)$$

To compare this result with the premature release requirement *R1*, which was represented in Section 2.3.2, we would need to find the maximum of the average value of P_2 in a one minute interval. This average may be evaluated from above by the maximum of P_2. P_2 reaches its maximum at time

$$t_0 = \frac{1}{\alpha}\cdot\ln\frac{\alpha+\mu}{\mu} \qquad (10)$$

The maximum of unavailability performance of the basic replication tool is:

$$\bar{A}_{b\text{-}max} = P_2(t_0) = \frac{\alpha}{\alpha+\mu}\left(1+\frac{\alpha}{\mu}\right)^{-\frac{\mu}{\alpha}} < \frac{\alpha}{\alpha+\mu}\frac{1}{e} < \frac{\alpha}{\mu}\frac{1}{e} \qquad (11)$$

Given that the average call holding time is 2 min, which means that $\mu = \frac{1}{2}$, and using the premature release requirement *R1*, we can obtain an estimate of the requirement for the mean time between failures of the basic replication tool. We will assume that the requirement *R1* should hold for *any* of the simultaneous calls in the system and that e.g. 10% of the requirement *R1* may be allocated to the replication scheme and unit changeovers may happen not more often than once in 20 min with constant distribution probability. We also assume that the probability of a unit changeover and the probability of a replication error are independent. Here the argument is not about the exact figures but the values obtained give a feeling of what we are dealing with.

$$\text{MTTF}_b \cong \text{MTBF}_b = \frac{1}{\alpha} > \frac{10^5}{2\,e}\frac{1}{\mu} = 613 \text{ h} \cong 26 \text{ days} \qquad (12)$$

The needed length of the experiment to yield a given number of allowed errors (n_e) can be calculated using the above formula. We will assume that the calls of the experiment form a serial system, the reliability of which can be calculated as a product of the reliabilities of the subsystems with independent failure modes and that the parameters of a field test: average call holding time and the number of parallel calls (N_c) to be handled by the test system are given. If less errors are found in the test, the system meets the reliability requirement.

The reliability of the parallel call system is $R_i = e^{-\alpha N_c \frac{1}{\mu}}$

Such parallel call systems again form a serial system the reliability of which is a product of the reliabilities of its components. We will denote the number of parallel call systems with *m*. Consequently, the reliability of the experimental call system is:

$$R = e^{-\alpha\, m\, N_c \frac{1}{\mu}} \qquad (13)$$

At the instant of the first error, this reliability should equal to e^{-1} . Thus, *m* may be calculated:

$$m = \frac{\mu}{\alpha\, N_c} = \frac{10^5}{2\, e\, N_c} \qquad (14)$$

So, the number of calls N yielding no more than 10 errors, should be:

$$N = n_e \cdot m \cdot N_c \cong \frac{10^5\ n_e}{2\, e} \cong 184\,000 \qquad (15)$$

Assuming a field trial with 5000 parallel one minute calls, the experiment should take about 37 min. This is the expected result on the basis of Eq. 12.

8.2 Reliability Model for the Schemes with Correction

A reliability model for a replicated call computation, when the corrective replication tools are also used, looks like the one in Figure 8.2. With η we have denoted the correction intensity which is the inverted value of the time for error detection and correction.

The set of simultaneous differential equations for this model is:

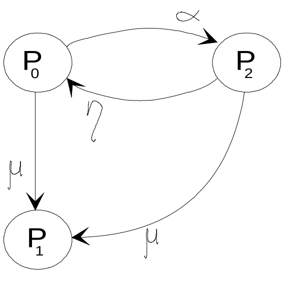


Figure 8.2 A Reliability Model of a Call, when Corrective Tools are used.

$$[P_0'(t), P_1'(t), P_2'(t)] = [P_0(t), P_1(t), P_2(t)] \text{ X } \begin{bmatrix} -(\alpha+\mu) & \mu & \alpha \\ 0 & 0 & 0 \\ \eta & \mu & -(\eta+\mu) \end{bmatrix} \tag{16}$$

As before we are interested in finding the solution to P_2 , which is obtained by taking the same initial values as before $P_0(0) = 1, P_1(0) = 0, P_2(0) = 0$:

$$P_2 = \frac{\alpha}{\eta+\alpha}\left(e^{-\mu t} - e^{-(\alpha+\mu+\eta)t}\right) \tag{17}$$

The unavailability performance of the model $\bar{A}_{c\text{-}max} = P_2(t_0)$ reaches its maximum at time

$$t_0 = \frac{1}{\alpha+\eta}\ln\left(1+\frac{\alpha+\eta}{\mu}\right) \tag{18}$$

The value of the maximum and an estimate for it from above are:

$$\bar{A}_{c\text{-}max} = \frac{\alpha}{\alpha+\mu+\eta}\left(1+\frac{\alpha+\eta}{\mu}\right)^{-\frac{\mu}{\alpha+\eta}} < \frac{\alpha}{\alpha+\mu+\eta}\frac{1}{e} \tag{19}$$

8.3 Analysis

Comparison of the estimates for the unavailability performance of the basic replication tool and the newly obtained value allow us to reason about the merits of the different corrective replication tools.

Relative merit of the corrective replication tool compared to the basic tool is expressed as:

$$M = \frac{\bar{A}_{\text{b-max}}}{\bar{A}_{\text{c-max}}} = \left(1+\frac{\eta}{\alpha+\mu}\right)\frac{\left(1+\frac{\alpha+\eta}{\mu}\right)^{\frac{\mu}{\alpha+\eta}}}{\left(1+\frac{\alpha}{\mu}\right)^{\frac{\mu}{\alpha}}} \tag{20}$$

An estimate for the merit is obtained by substituting the maximums with their estimates:

$$M \cong 1+\frac{\eta}{\alpha+\mu} \cong 1+\frac{\eta}{\mu} \tag{21}$$

because it is assumed that $\alpha \ll \mu$.

Using our measure for the merit of different replication schemes we are able to draw some qualitative conclusions. (i) If a corrective replication tool, like our basic error correction tool, is able to detect and correct errors in one tenth of the average call holding time, this tool will improve the reliability of the replication scheme about eleven times. A faster tool, which detects and corrects errors in one hundredth of the call holding time, will improve the reliability of the scheme by about one hundred times.

(ii) Better performance of the replication scheme, when an error correction function is included, can also be used to cope with more frequent detectable errors which lead to an inconsistent state of the spare computation and to meet the premature release requirement at the same time. So, in this case the requirement for the MTBF of the basic tool may be loosened.

(iii) Our result allows us to conclude that one of the correction methods, the *time-out triggered synchronisation is not an effective tool* because it gives an improvement in reliability using our measure for the merit of the tool by only *a few tens of percents*, taking that the triggering time-outs have to be at least two times the average call holding time for performance reasons.

Although we have not shown the modelling results for the case when there are some undetectable errors in the performance of the basic scheme, we may formulate our expectations for this case also, by looking at the above two models together.

(iv) If there are some undetectable errors in the performance of the basic model, the achievable improvement in reliability of the scheme by correction tools is restricted by the portion of undetectable errors in the total amount of errors. So, in this situation the fraction of undetectable errors sets a limit to the reasonable performance of the correction tool for detectable errors.

(v) What can we expect by applying the basic error correction tool described in Section 7.3? To answer this question, at least qualitatively, without an implementation and a tedious field trial with real applications, we need an estimate or at least an educated guess at the correction time. The correction CPU time (T_c) for this tool is the sum of the error detection time (t_d), the execution of error analysis (t_a), and the execution of the correction algorithm (t_{ca}), i.e.:

$$T_c = t_d + t_a + t_{ca} \tag{22}$$

The last two (t_a and t_{ca}) can be expected to be less than 100 ms, while t_d may be much longer and is hard to determine without experimental data. Heuristic reasoning gives us a guess at t_d. Some errors in the kernel state of a process (error type A in Section 7.1) are detected quickly in times which are comparable to the computation skew between the spare and the active unit (error type A.1 in Section 7.2) or to kernel time-outs (error types A.5, A.6 in Section 7.2). However, we can hardly expect that these types of errors are typical because they probably occur in case of message loss errors. Instead, we expect that message ordering errors are more typical. In such cases, we will assume that the error manifests itself to the application state (error type B in Section 7.2). If an application state is connected to waiting for an internal message from another application process, we may assume that the time-out is less than a few seconds. If the process was waiting for a message from the network, which is a minority of the messages, the time-outs are usually tens of seconds. On the basis of this rather vague reasoning, we make a guess that the average t_d = 20 sec.

Consequently, under this assumption and assuming that great majority of all the relevant errors are detectable, we get: $\eta = 3$ and the relative merit of the basic error correction tool, when applied to call computations, can be expected to be $M = 7$.

(vi) What can we expect from the asynchronous error correction tool? We will consider only those application states to which the tool is applied. In other states the errors are assumed to be unimportant. The error detection time t_d is comparable to the length of the time-out T_{cs} which is set when one of the checksums arrive and has to be taken to cover for the computation skew under heavy load conditions multiplied by a small integer. The computation skew can be expected to be no more than a few hundreds of milliseconds. For the correction time the formula applies:

$$T_c = t_d + t_a + t_{ca} = \mathrm{k} \cdot \Delta r + t_a + t_{ca} \tag{23}$$

Consequently, we may assume that T_c is of the order of one second. This gives $\eta = 60$ and the merit of the tool compared to the basic replication tool is of the order of one hundred, provided that correction succeeds for individual warm-up entities with high probability and the correction order meets the requirement of Lemma 7.1.

9 Language Issues

In this chapter we shall discuss the possibilities to hide the details of the replication related run-time system primitives from the application programmers and thus reach a high level of transparency of the replication scheme using a language approach. Language tool support for the use of replication primitives aims at handling the replication issues of the applications as a programming-in-the-large effort. This means that the language tools should be able to express, in a compact form, the system design decisions related to the replication of computations.

Talking on a high level of abstraction, two approaches can be suggested to support replication of computations in a language. The first is to define language constructs corresponding to each aspect of the replication primitives. Clearly this approach leads to quite a tedious programming effort because there may be a large number of process types and message types in the applications to be dealt with. The second approach suggests defining a small number of high level abstractions from which the default treatment of the run-time replication primitives is derived. A combination of the approaches is also possible allowing the changing of the default treatment of the primitives in desired special cases while most events are handled as derived from the high level abstractions.

We will represent the language constructs as production rules of a language, like TNSDL [TNG]. The left hand side of a rule is separated from the right hand side by "::=". Non-terminal symbols are placed in parenthesis "<", ">". Terminal symbols are written in capital bold, e.g. **OF**. Rules end in a semi-colon (;). Lists are represented by using recursive rules. A vertical bar (|) stands for an alternative. We will not go into details of the possible language tool implementation nor will we exactly follow the representation rules used in the TNSDL grammar [TNG]. Our intention is limited to showing the possibilities of language tools in helping the replication design work and setting the requirements for tool implementation.

The language tools should be able to produce the following:

1. delivery codes of the message types,
2. replication related message attributes:
   - discard message if sender is a non-active computer,
   - conditional replication of a message during the warm-up of the spare unit if the "replication_mode_ok" of the sender is false and the "replication_mode_ok" of the receiver is true,
   - correction request attributes,
3. warm-up requests to warm-up control on the basis of the definition of the warm-up entity classes.

We will now show how some of the above objectives can be reached using the second approach. The presentation will rely on our target language, TNSDL. Useful abstractions for our purpose are *replication class* and *process attribute*.

For the presentation to be self-contained a few features of TNSDL are needed. Systems described in TNSDL are decomposed into system blocks, system blocks into service blocks, and finally the service blocks into program blocks. Declarations may reside on any of these levels while all the code resides in the program blocks. In program blocks there may be process families and modules. Usual visibility rules of block structured languages apply to declarations of the hierarchical blocks, families and modules. A block has an interface and an implementation The implementation may contain references to lower level blocks and on the program block level to process families and modules. The TNSDL block structure and other main features have been explained e.g. in [Sea91].

9.1 Replication class declarations

The replication class concept was introduced in Chapter 5. Now we will define it as a language construct:

<Replication class definition> ::= **REPLICATION_CLASS** <Replication class name>;

<Replication class name> ::= <name>;

Names of the replication classes have to be defined on one of the higher levels of block hierarchy in a block structured language, like TNSDL. The usual scope rules also apply to the replication class names. Processes can be attached to replication classes in the program block implementation as follows:

<programBlockImplDeclaration> ::= <process references>| <otherDefs> |<Replication definitions>;

<Replication definitions > ::=< process typeDef list>;

<process typeDef list> ::= <process typeDef list>**,**<process typeDef>|<process typeDef>;

The first rule shows how the replication definitions can be tied to the present TNSDL grammar [TNG]. Replication definitions reside after the process decomposition of the program block has been declared. Process attributes are qualifiers of process type which e.g. define the outlines of how the run-time replication primitives are applied in conjunction with a process of that type. We suggest to define one process attribute for this purpose. We will call it the *Replication mode* attribute and it has four values, namely *tightly modular*, *loosely modular*, *primary/standby* and *cold*.

A short informal interpretation and some examples of the use of values of the Replication mode process attribute are useful. TIGHTLY_MODULAR is intended to be used e.g. for hand processes in applications like call control and signalling. The LOOSELY_MODULAR scheme behaves as TIGHTLY_MODULAR relative to the basic replication scheme. The difference is that automatic correction propagation is not used towards LOOSELY_MODULAR processes. Example candidates of LOOSELY_MODULAR processes are some statistical data collection processes in the DX 200 system. The PRIMARY_STANDBY attribute value can be used e.g. for master processes to cope with the hand process allocation scheme which was presented in Section 5.1.3. The COLD attribute value can be used e.g. for database service provider processes which offer stateless services to call control applications.

The language constructs are:

<process typeDef> ::= <process type name> **OF** <Replication class name> **IS** <process attribute>;

<process type name> ::= <name>;

<process attribute> ::= <Replication mode>;

<Replication mode> ::= TIGHTLY_MODULAR | LOOSELY_MODULAR | PRIMARY_STANDBY |
COLD;

How exactly TIGHTLY_MODULAR, LOOSELY_MODULAR, PRIMARY_STANDBY and COLD are defined is a translator implementation issue but for example they can be defined as keywords or as constants at the highest block level of a TNSDL system. From the translators point of view the last rule then would actually read:

<Replication mode> ::= <enum[1:4]>.

Default values for message attributes and delivery codes can be produced from these constructs applying a set of rules. In the rules we will denote p - sender process type, q - recipient process type, RC, RC1, RC2 - variables the values of which are replication class names. By $\in$ we will denote the relation of belonging to a replication class.

The rules are:

```
if q is tightly_modular or q is loosely_modular
    q is modular
if $p \in RC$ and $q \in RC$ and q is modular
    delivery_code = own_computer
    conditional_replication_attribute = true
    destroy_in_non-active_unit = false
if $p \in RC$ and $q \in RC$ and q is primary_standby
    delivery_code = own_computer
    conditional_replication_attribute = false
    destroy_in_non-active_unit = true
if $p \in RC$ and $q \in RC$ and q is cold
    if p is cold or p is primary_standby
        delivery_code = own_computer
        conditional_replication_attribute = false
        destroy_in_non-active_unit = true
    else
        not allowed /* warning */
if $p \in RC1$ and $q \in RC2$ and $RC1 \neq RC2$ and q IS MODULAR
    delivery_code = WO+SP
    conditional_replication_attribute = true /*don´t care */
    destroy_in_non-active_unit = true
if $p \in RC1$ and $q \in RC2$ and $RC1 \neq RC2$ and (q IS PRIMARY_STANDBY or q IS COLD)
    delivery_code = WO
    conditional_replication_attribute = false
    destroy_in_non-active_unit = true
```

To implement these rules, one way or the other, the process attributes of the receiver have to be available, when a send is executed. A possible way of doing this is to generate a global process attribute table which is made directly available in all the computer units along the message bus in our computing system. In this table an entry is needed for each process type. Then either these rules could be used to generate code which is executed by the application before each send or the run-time system could provide an enhanced send service.

Correction propagation requests can be handled assuming that the initial correction request is defined as a constant attribute value in a message declaration and that the run-time system sets the correction request attribute for the receiving process when a message with a correction request attribute is received. This run-time process attribute is set to false at the end of the transition. Then the language tool may use the following generation rule:

```
if p sends m to q and q IS TIGHTLY_MODULAR
    if correction request attribute of p is true
        set correction request attribute of m
```

If a choice between more than one correction method (e.g. the asynchronous error correction tool and safe transitions) has to be made upon reception of a message with the correction request attribute, this can be handled by defining a new value of the Replication mode process attribute corresponding to each correction method. All these new values are refinements of the above TIGHTLY_MODULAR replication mode. The run-time system can then make the choice between the methods of correction by reading the value of the refined replication mode process attribute.

9.2 Warm-up declarations

Tedious details of warm-up requests can be hidden from the application programmers by defining a few additional qualifiers to supplement some of the language constructs of the TNSDL language which were initially defined for non-replicated applications. Warm-up entity classes may contain a number of data components declared as variables of a process or a module. Only modules which belong to a program block also containing a process family may have variables which have to be warmed up. Such variables are warmed up with the master process of the family. The data components are:

a. statically allocated variables

b. dynamically allocated buffers

c. files or sets of data units of a file

Case (a) is simple because variables may be handled independently. In case (b) the buffer handle data structure, buffer length and possibly a variable pointing inside the buffer have to be bound together. In case (c) the warm-up method of the file, the possible decomposition of the file into (identical) data units and the file number have to be declared.

To meet these needs the required language constructs are:

<variableDefinition> **::= DCL** <variableModifier> <variableDeclarationList> end;

<variableModifier> **::= WITHWARMING** | **SAVE** |<>;

SAVE is a qualifier used in procedure variable declarations and is of interest here only to tie up the presentation to the initial TNSDL grammar [TNG].

<variableDeclarationList> ::= <variableDeclarationList>**,** <variableDeclaration>|
<variableDeclaration>;

<variableDeclaration> ::= <nonemptyVariableList> <typeName>

| <VariableName> <typeName>**(**<TypeModifierList>**)**

| <other types of variableDeclarations>;

<TypeModifierList> ::= <qualifier1>**,**<qualifier2>|<qualifier1>**,**<qualifier2>**,**<qualifier3>;

<qualifier1> ::= BUFFER | FILE_NOT_WARMED | FILE_WARMED_ENTIRELY |
FILE_SLICED_STATICALLY | FILE_LOADED_WITH_PROCESS_WARM-UP |
FILE_WARMED_CONTINUOUSLY ;

<qualifier2> ::= <length>| <fileNumber>;

<qualifier3>::= <farPointer_to_buffer> |<fileSliceVariable>;

How BUFFER, FILE_NOT_WARMED etc. are defined is a translator implementation issue but for example they can be keywords or constants declared at the system level. So, again in the last case the rule for qualifier1 would then actually read:

<qualifier1> ::= enum[1:6];

It is assumed that Type Modifier Lists may be bound to variables of Buffer_Handle type and File_Handle type. Qualifier2 may carry the length of a buffer if the value of qualifier1 is BUFFER or a file number in case of other values of qualifier1. Qualifier3 may carry the name of a long pointer declared in an earlier declaration statement without the "**WITHWARMING**" modifier, pointing to the buffer. Qualifier3 may also carry a variable of predefined type "slice_info_type", which has been declared earlier and initialised in the START transition of the process. It is assumed that this variable defines the data unit in a file to be warmed up with the process.

Cases for files are

FILE_NOT_WARMED only the file handle is warmed up,

FILE_WARMED_ENTIRELY the whole file is warmed up together with the process in an atomic action, FILE_SLICED_STATICALLY a data unit of the file defined by qualifier3 is warmed up with the process in an atomic action. Note that only in this case the Slice Variable is needed,

FILE_WARMED_CONTINUOUSLY the file is warmed up separately from the process using the algorithm for a set of separate data units in the spare unit defined in Chapter 6.

FILE_LOADED_WITH_PROCESS_WARM-UP
the file is loaded into the spare unit while the concerned process is frozen just after the process was warmed up.

The translator has to keep all the variables to be warmed up in a continuous data structure to meet the warm-up request interface specification. It should be possible for the translator to ignore the warm-up qualifiers if at translation time this is desired. When the warm-up qualifiers are taken into account the translator produces the warm-up request code of the master process. For each hand process type, the translator produces a function returning a data structure which carries information about the data components to be warmed up in conjunction with the hand process type. The master process can then call these functions.

The described warm-up support has been implemented in the TNSDL translator.

An alternative implementation approach would be to generate a data table from the warm-up qualifiers and augment the executable code with this table. Then the run-time tools would have to understand the structure of the table and how to find it in a program block executable image. This approach seems cleaner, but at the time when these features were first used, had the practical disadvantage of binding together tool development with the system development in a project possibly with tight time schedules.

10 Evaluation of the Solution

10.1 Performance Cost of Replicated Computations

From the beginning we have been looking for a solution to the problem of replicated computations which would bear no more than a non-significant performance overhead compared to the case of non-replicated computations. One of the reasons why we did not start our replication design with an atomic broadcast protocol was the expected performance penalty. Did we meet our initial goal for our target applications? We will use the message count measure to answer this question qualitatively. For clarity we define non-significant performance overhead as a few tens of percents (20-30%) overhead by the message count measure.

Let us take an example of a process configuration handling a call. The example is close enough to how call processing is done in the DX 200 system to be representative for our target system. In the configuration there are 3 state oriented processes in the incoming (A) unit and two processes in the outgoing (B) unit. The whole message budget for the non-replicated call computation is 200 messages. One of the units (let us assume A) is a 2*N* redundant unit while the other is *N*+1 redundant.

When the computation is replicated (in the 2*N* unit) using the basic scheme, message cost penalty for this under the no-failure assumption is:

- two messages/process allocation
- two messages/process release
- no time-out synchronisations because time-outs are only set and deleted

The cost is: C_1 = 12 messages, which is a 6% addition to the total message budget. Note that we assumed that wo+sp deliveries bear no performance penalty because the multicast is supported by hardware and it is assumed that all the state oriented processes working on the same call in a computer unit belong to the same replication group.

What is the message cost penalty for changing all the transitions into safe transitions? For the calculation, we have to know the number of messages received by the three 2*N* replicated processes (we will denote this number S). Bearing in mind that there are a number of other processes involved in handling the call which communicate with the processes of the state oriented execution group mentioned above, we will assume that S = 60.

The cost penalty under the no-failure assumption is:

- two external messages/received message with the correction request attribute,
- two internal messages/received message with the correction request attribute.

Cost $C_2 = 4 \times 60 = 240$ messages, which is significant. The overall message budget of a call is more than doubled, and even if we ignore the internal messages in the safe synchronisation algorithm, the increase is 60%, which is also unacceptable. But if we add the correction requests only to the carefully selected important events, the performance cost is much lower. The cost per important external event is:

$c_i = 3 \times 4 = 12$ messages, which represents a 6% increase to the overall message budget.

What level of performance penalty should we expect if we would use a reliable multicast protocol on the basic level of the replication scheme? As an example we will take the rel/REL_{fifo} protocol having the Fifo Multicast property suggested in [Ez91]. The protocol requires the message being sent twice and each of them being acknowledged, so the message count is quadrupled if we have one sender and a paired receiver, and there are no failures, which is assumed to be the representative case. Optimisation of the acknowledgements seems possible, when there happen to be suitable messages available to also carry along the acknowledgements. We assume that such suitable messages could be in our case only application level acknowledgements. If application level acknowledgements are assumed, then the low level *rel* -protocol acknowledgements are excessive anyway from a global perspective. So for fair comparison, we should assume that the protocol is on its own.

Consequently, starting to build the replication scheme from reliable atomic broadcast protocol, we should expect that:

- all internal messages of a computer unit are substituted by external messages,
- the message count per one call computation would be quadrupled or, if optimisation is possible, at least doubled.

In any case, clearly the performance overhead by message count measure would not be non-significant and would not leave space for the possible additional messages required by the higher layers of the replication scheme in the total message budget. Because all the messages would be delivered through the message bus, a performance bottle-neck would be created in the message bus interfaces.

Consequently, assuming that our example process configuration is representative, we can see that the *performance penalty* for the replication of call computations *is non-significant* if the suggested *basic replication scheme is used* and on top of that *no more than two or three external events are handled by safe transitions* propagated through all the state oriented replicated processes. This is enough to exclude the need to allocate some portion of the premature release requirement *R1* defined in Section 2.3.2 to the replication scheme. Any alternative hypothetical replication scheme built on top of a reliable atomic multicast protocol should be expected to bear a significant performance penalty.

10.2 Design Rules Imposed on the Applications

In Section 6.6 we already discussed some useful design rules that should be followed by replicated applications to comply with the requirements of the warm-up services. In Section 5.2.1 we introduced the replication group concept. We expressed our hope that all the state oriented processes of a computation could reside in one replication group. However, we did not try to identify any restrictions to this "rule".

Let us look at an example of a call computation of the previous section. If the data identifying the circuits used by the call would be inconsistent in the spare and the active process, this could lead the computer unit to mix up the circuits of the calls it handles after unit changeover. Such an error would be possible if data about free circuits from the spare side was used by the state oriented processes on the spare side. This could not happen if the process executing the search algorithm and the users of this data are put in separate replication groups. This is an example of how events carrying important information may influence the membership of processes in replication groups and restrict the size of the replication groups, e.g. in the DX 210 configuration.

Another example is when the charging is started by one of the state oriented processes on the incoming side. We want that this important decision on the CONNECT message from the outgoing side is taken simultaneously by both the active and the spare process. To a certain extent this can be helped by putting the incoming side processes and the outgoing side processes in separate replication groups even in the DX 210 configuration. In a large configuration there would always be two separate replication groups because the processes may and usually do reside in different computers.

An heuristic generalisation of the previous examples is: If we want to *restrict error cascading* in the spare computation through a chain of processes or want to *minimise the computation skew* of some of the replicated transitions in the computation, it is recommended that the process chain is broken down by a replication group boundary. The boundary is put next to the source of the potential error and next to the process hosting the transition which we want to start without computation skew.

10.3 Transparency of the Replication Scheme

The suggested replication scheme is not fully transparent. However, application experience of the first users of the replication primitives including the basic scheme and the warm-up services has shown that when the requirements imposed by the replication scheme on the application are understood, changing an initially non-replicated code into a replicated one using also the active warm-up services takes only from a few hours to a couple of days even while the full TNSDL support was not implemented yet. Additionally, some amount of testing is necessary. This is a vast improvement over the previous situation where the replication issues had to be handled without any kernel support by the applications. Then the same work could take months. There are additional savings in the efforts for the specification phase because the replication scheme requires a set of design rules to be followed. In such a situation you do not tend to invent the wheel again. The achieved level of transparency also means that if need be the replication scheme or its implementation could be changed without drastically influencing the applications as long as the new model or implementation are at least as transparent to the applications as the present one. So, we can say that the modularity of the whole system has been improved by the replication support of the kernel.

To finalise this issue, we will make a list of visible features of the replication scheme from the designer's point of view:

- system design effort is required to group the processes into replication groups for the sake of performance optimisation and to make sure that the application is structured in such a way that the system does not become too complex for the warm-up to handle,
- description of the replication groups is made using the TNSDL features described in Chapter 9,
- time-outs to be synchronised from the active to the spare are selected and the actual parameters of the calls for setting the time-outs are checked,

- the data components to be warmed up in connection with a process type are chosen and their description is augmented by the required warm-up qualifiers,
- whether some particular warm-up order is required for the processes or not is checked. If the answer is yes, this information is written into a new version of the constant data tables describing the warm-up order,
- possible important events are chosen and the correction request attributes are set into the corresponding messages,
- a software build is made, system test is set up and performed.

10.4 Contribution to Knowledge

The basic replication scheme, the warm-up algorithms, the warm-up data qualifiers in the TNSDL language and the basic error correction tool have been implemented in the target system. There are also a number of public network and GSM application programs which use these tools.

To our knowledge this is the first paper on software supported replicated computations in a distributed switching environment. This is why we used a lot of space discussing the basic requirements to be met by replicated computations in a switching system. To ensure that we stay strictly in the realm of real world problems and for the information of those who are not experts on switching, the replication related features of the target environment were described in Chapter 1. In Chapters 2 and 3 we saw that the relevant requirements on replicated computations are different from the requirements which have been tackled in other application areas of fault-tolerant computing, like long-life systems, life-critical systems and transaction processing systems. Probably this is why we did not find a satisfactory replication solution in the extensive literature written on the replicated computations in those other application areas. So, we came to the conclusion that a new switching oriented replication solution was needed. In Chapter 4 and sections 5.1, 5.2, 6.2, and 7.1 we introduced modelling tools to help us to discuss the properties of the replication scheme. The model of computation was based on the Asynchronous Communication Tree model supplemented with data units and their values. Information can be passed from one process to another in messages as well as through read and write actions to the data units. In Chapters 5, 6 and 7 we described the replication scheme which was structured as a set of tools.

The model of computations is presented in Definition 4.8 augmented by definitions 5.7 and 5.8. Definition 5.3 defines what we mean by a replicated computation.

The approach of starting from an eventual convergence model as opposed to aiming at instantaneous correctness at the basic level of the model has been identified before[e.g. Co85]. Examples of true development of such an approach are to our knowledge, however, non-existent.

In Section 5.2.1 the replication group concept was introduced. This allows the replication of larger objects than processes. The consequent properties of the scheme were discussed in Section 5.4 and Lemmas 5.9, 5.10 and 7.1. It was concluded that the replication group concept is an optimisation tool having a trade-off between reliability and performance. To our knowledge such a feature has not been incorporated into any of the replication models known so far. This is not surprising because such a concept would obviously not fit well together with a replication scheme aiming at instantaneous correctness. In the current implementation, this tool is available but is not in particularly wide use due to message ordering problems predicted in the theory and due to the abundance of performance available with the current hardware. Internal messages are, however, used in specific situations where this does not increase the probability of an error.

In Chapter 6 a thorough analysis of necessary and sufficient conditions of success of migration of computations was made. The challenge of the task was based on the fact that message passing is not the only means of sending information from one process to another. However, based on our model of computations we managed to find a solution covering the warm-up of processes and data units. The results of the analysis were implemented in sections 6.3 and 6.4 by two real-time algorithms for migration or warm-up of computations without roll-back features. By this we mean that none of the application transitions are re-executed by the spare computer to catch-up with the active unit. Due to the fact that roll-back is not needed, the algorithms are well suited to the real-time requirements of switching. To our knowledge, such algorithms have not been described before. These algorithms are based on the concepts of warm-up entity (Definition 6.8), warm-up entity class and warm-up order (Definition 6.10). To our knowledge, such a set of concepts is also a new item which has not been developed before.

In Chapter 7, several corrective replication tools augmenting the basic replication tool were introduced. Much attention was given to the safe transition tool. The consistency properties of this tool were formulated by using the modelling tools of Chapter 4, Lemma 7.1 and Definition 7.2. The safe transition tool allows the addition of conditional instantaneous correctness[25] to a replicated computation in conjunction with selected important events.

In modelling the semantics of computation, the work done in [Lin91] on using Asynchronous Communication Trees (ACT) was taken as a basis. The model of [Lin91] was initially supplemented with data units as an alternative of message passing in information exchange between processes (Definition 4.8). The data units were introduced as an abstraction of the memory resident files which are extensively used in the target system. Later in Chapter 5 the model of computations was augmented with the creation and deletion of processes and used to model replicated computations (Definitions 5.7 and 5.8). In Chapter 6, due to the model of computations, we were able to formally define how the atomic warm-up entities can be constructed (Definition 6.8) as well as to introduce the warm-up order (Definition 6.10). In Chapter 7, Lemma 7.1 due to the model of computations, we were able to show how replication errors may propagate in a system of processes and data units and under what conditions restricted correction of the state of a spare computer unit, infected by a replication error, can be successful.

In Chapter 8, the well known Markov's reliability modelling techniques were used to investigate the properties of the basic replication scheme and to compare different corrective replication tools. From the model of a replicated call computation, by comparing the performance of the replication tool with the premature release requirement for established calls (*R1* in Section 2.3.2) we obtained a qualitative estimate of the requirement for the mean time between failures of the basic replication tool. The tool has to meet this requirement to be applicable as such to call computations.

On the basis of an educated guess, the basic error correction tool was found to give a decrease in unreliability of the replication scheme by 7 times for call computations. The asynchronous error correction tool was found to perform better, giving a decrease of unreliability of the replication scheme compared to the basic scheme on the order of a hundred times for call computations. The comparison further allowed us to discard one of the tools suggested in Chapter 7, namely the time-out triggered synchronisation.

[25] Definition 7.2.

In Chapter 9, the possibilities of language tools to enhance the transparency of the replication scheme to the application programs were discussed. With the suggested language features, the replication design decisions that have to be made, can be incorporated into the applications in a declarative fashion.

Two questions have to be asked. First, how target environment dependent are our results? We claim that at least the above mentioned new items to knowledge are *generally attractive* features of replicated computations in any distributed switching system aiming to meet the availability and reliability requirements for the public switches set by the CCITT and offering convenient capabilities to the operator to reallocate the processing resources of the system. This does not exclude the possibility of finding other valid solutions to meet those requirements. The results may also be of interest in other possible application areas where systems with low levels of redundancy (N+M, where N is the number of active machines, M the number of spares and $M < N$) are used.

Second, does the developed replication scheme meet the requirements and design goals set at the beginning of this thesis? Our reasoning has been that the basic scheme gives us a certain grade of service and if the achieved grade of service is not enough, we have shown that with the corrective tools it can be improved to the level required. So, we hope that this last chapter has convinced the reader that the answer is a definite yes.

In fact the (not formally collected) field experience so far indicates that the basic replication scheme, which has been applied to several types of applications except call control and signalling, has not caused any real trouble and is performing better than expected by the kernel development team.

10.5 Further development

The wider range of applications, using the replication tools, than originally anticipated raises requirements for further development of the replication tools as well as calls for improvements in the design practice to ensure compliance to the design rules and recommendations imposed by the replication tools. The improvements in the design practice could be supported by improvements in the design tools. These include the full range of TNSDL support for the description of replication groups presented in Chapter 9 and better debugging tools.

Improvements in the implementation of the run-time replication tools that have now been identified include:

- an optimised and safer implementation of the hand allocation protocol well suited for TNSDL applications,
- a performance optimised variant of the hand release procedure,
- a performance optimised warm-up by skipping the warm-up of free hands,
- better phasing of the overall warm-up by announcing the spare state only after the master processes have been warmed-up.

A customer requirement addressing the original application scope of the run-time replication tools has now been raised: a graceful unit changeover initiated by an operator command should be clean. With the current implementation most times at least one error is indicated. These errors are of the type expected in this study; e.g. a call in the set-up phase is lost or some trouble is caused by the failure of the simple multi-cast protocol as discussed in Section 5.3.1. This particular customer requirement, as well as the needs of added reliability of a wide range of applications other than call control and signalling, could be addressed by the implementation of the safe transition tool presented in Chapter 7. Additional work is needed on the multi-cast protocol either to eliminate or avoid the loss and duplication of messages during the unit changeover. The severe performance requirements identified in this study also apply to any improvements of the multicast protocol, provided that such improvements are meant for general and frequent use by applications which are critical in terms of system performance.

10.6 Conclusion

In general, there are very few academic technical papers on fault-tolerance of switching systems. We hope that this thesis inspires others to make contributions in this area and fill the obvious gap. From my part I can say that writing this thesis has been a useful exercise; it has contributed to the identification of problems and the solution of new issues which were raised during the introduction of the replication tools into the target system. It has also helped to understand how, why and why not some existing solutions work and what can be done to improve them.

Our track record shows that the replication services themselves have contributed to the rapid implementation of e.g. replicated call control, signalling and statistics collection applications although the debugging of such applications is still felt to take too long. An example of applications using the described run-time and the corresponding design tools are Nokia's GSM network elements based on the DX 200. Where bigger suppliers had considerable difficulties Nokia succeeded in the timely introduction of the whole range of GSM network elements partially due to the language and replication tools which we have described in this thesis.

The experience is that the effort put into the design of the replication scheme has certainly been worthwhile and has already paid off in the DX 200 environment during the first couple initial years of application.

REFERENCES

[Af] Aflatuni, Availability Performance of Digital Switching System DX 200, Telenokia Specification CAN 2038.

[Ag] Gul Agha, Supporting Multiparadigm Programming on Actor Architectures, Department of Computer Science Yale University, New Haven, Connecticut, USA.

[Ag86] G. Agha, Actors: A Model of Concurrent Computation in Distributed Systems, MIT Press, Cambridge, Mass., 1986.

[Ar89] M. Arppe, E. Hartikainen, R. Kantola, A. Muittari DX 200 F5 Lämmityksen periaatteet, Telenokia company document 1989.

[Ar90] The Arjuna System, Selected Papers 1987-90 Computing Laboratory University of Newcastle upon Tyne, Claremont Tower, Claremont Road, Newcastle Upon Tyne, NE1 7RU.

[Ba81] Joel F. Bartlett, A NonStop Kernel in Proceedings of the Eighth Symposium on Operating System Principles, Dec. 1981, pp. 22-29 by ACM. Also in John A. Stankovic (ed.) Reliable Distributed System Software IEEE Computer Society press, 1985.

[Bir] P. Birman, T.A. Joseph, Exploiting replication in distributed systems Ch. 15 in Sape Mullender (ed.) Distributed Systems ACM Press, New York, Addison-Wesley Publishing Company.

[Bi85] P. Birman, Replication and Fault-Tolerance in the ISIS System. In Proceedings of the Tenth Symposium on Operating System Principles, pp79-86, Orcas Island, Washington, Dec. 1985.

[Bl90] David L. Black, Scheduling Support for Concurrency and Parallelism in the Mach Operating System, IEEE Computer Vol. 23, No 5 May 1990.

[Bo87] Anita Borg, Fault Tolerance by Design, Unix Review, April 1987.

[Ch88] R. Cheriton, The V Distributed System, In Communications of the ACM Vol 31, No 3 pp.314-333, March 1988.

[Cl86] Clement, P. K. Giloth, Evolution of Fault Tolerant Switching Systems in AT&T. In Evolution of Fault-Tolerant Computing. In Honor of William C. Carter, Baden, Austria, 30 June 1986.

[Co85] Eric Charles Cooper, Replicated Distributed Programs Ph.D. dissertation, Report No UCB/CSD 85/231 May 1985 PROGRES Report No. 85.5 Computer Science Division (EECS) University of California Berkeley, California 94720.

[Da88] Dasgupta, R. J. LeBlanc Jr., W.F. Appelbe The Clouds Distributed Operating System, The 8th International Conference on Distributed Computing Systems, IEEE, June 1988.

[Ez91] Paul D. Ezhilchelvan, Santosh Shrivastava A Distributed Systems Architecture Supporting High Availability and Reliability, Computing Laboratory, University of Newcastle upon Tyne, England, UK.

[Gl84] Glazer, Fault Tolerant Mini Needs Enhanced Operating System Computer Design, August 1984. Pennwell Publishing Company. Also in Victor P. Nelson and Bill D. Carroll Tutorial: Fault Tolerant Computing IEEE Computer Society press.

[Gr81] David Gries, The Science of Programming, Springer Verlag, 1981.

[Jo87] D.Johnson, W. Zwaenepoel, Sender-Based Message Logging. In Proceedings of the 17th International Symposium on Fault-Tolerant Computing pp.14-19, Pittsburgh, Pennsylvania July 1987.

[Ka87] Raimo Kantola, DX 200 F5, Varmennuksen periaatteet, internal Telenokia document, 1987.

[Ka88] Raimo Kantola, DX 200 F5, Varmennuksen periaatteet, internal Telenokia document, 1988.

[LA89] Ari Lehtoranta Puhelinkeskuksen DX 200 käytönohjaustietokoneen varmentaminen, 1989. Diplomityö, HTKK, Sähkötekniikan osasto.

[Lin91] Augmenting SDL specifications with LOTOS behaviour expressions, Markus Lindqvist and Heikki Tuominen, in Software Engineering Environments, vol. 3.

[Ma89] Mancini, S. K. Shrivastava, Replication within Atomic Actions and Conversations: A Case Study in Fault-Tolerance Duality. In Proceedings of the 19th International Symposium on Fault-Tolerant Computing pp. 454-461, 21-23 June 1989 Chicago, Illinois.

[Mo89] Michele Morganti, F-T in Telecommunications Networks: State, Perspectives, Trends. In Proceedings of the 19th International Symposium on Fault-Tolerant Computing pp. 253-258, 21-23 June 1989 Chicago, Illinois.

[Mu90] Sape J., Mullender, Guido van Rossum, A. S. Tanenbaum, etc. Amoeba: A Distributed Operating System for the 1990s. Computer Vol. 23, No 5 May 1990.

[Mu86] Mullender, A.S. Tanenbaum, The Design of a Capability-Based Distributed Operating System, In The Computer Journal, Vol 29, No 4, 1986 pp. 289-300.

[NaTa] Natarajan, J. Tang, Kernel Mechanisms for Distributed Real-time Programs, Seventh Annual Int. Phoenix Conference on Computers and Communications, March 1988.

[Ne87] Victor P. Nelson and Bill D. Carroll, Tutorial: Fault Tolerant Computing IEEE Computer Society press, 1987.

[Ne56] J. von Neumann, Probabilictic logics and the synthesis of reliable organisms from unreliable components, In Automata Studies, ed. by C. E. Shannon and J. McCarthy, Princeton University Press, 1956, pp 43-98.

[Pe80] M.Pease, R. Shostak, L. Lamport. Reaching agreement in the presence of faults. Journal of the ACM 27(2), April 1980, pp228-234.

[PuAf] T. Purho, Z. Aflatuni, J. Soitinaho Operational Reliability of the DX 200 Switching System, Proceedings of the Annual Reliability and Maintainability Symposium, IEEE, 1987.

[Q.7XX] Blue Book -Fascicle VI.7 Spesification of Signalling System No 7. Recommendations Q.701 - Q716.

[Q.543] CCITT, Blue Book. Recommendation Q.543.

[Re89] Robbert van Renesse, A. S. Tanenbaum, S.J. Mullender, The Evolution of a Distributed Operating System. In W.Schröder-Preikschat, W.Zimmer (Eds.), Progress in Distributed Operating Systems and Distributed Systems Management European Workshop, Berlin, FRG, April 18/19, 1989 Proceedings Pub. in Lecture Notes in Computer Science Springer-Verlag.

[Re84] Rennels, Fault-Tolerant Computing - Concepts and Examples IEEE Transactions on Computers, Vol. C-33, No 12 Dec. 1984, pp.1116-1129. Also in Victor P. Nelson and Bill D. Carroll Tutorial: Fault Tolerant Computing IEEE Computer Society press.

[Sc83] R.D. Schlichting, F. B. Schneider, Fail-stop processors: An approach to designing fault-tolerant computing systems. ACM Transactions on Computer Systems 1(3), August 1983, pp 222-238.

[Sc84] F.B. Schneider, Byzantine generals in action: Implementing fail-stop processors. ACM Transactions on Computer Systems 2(2), May 1984, pp145-154.

[Sea91] E. Kettunen, M. Lindqvist, E. Ruohtula, H. Tuominen, A Seamless Software Production Process Based on TNSDL, Nokia Telecommunications, Proceedings of the TELECOM-91, Geneva 1991.

[Si89] Jon Silverman, T.Raeuchle, H. Madduri, Programming Fault-Tolerant Distributed Applications in HOPS, Proceedings of the Eighth Annual International Phoenix Conference on Computers and Communications March 1989.

[Sn84] Snead, Frank Ho, B. Engram Operating System Features Real Time and Fault Tolerance, Computer Design August 1984.

[Sp89] Speirs, P. A. Barrett, Using Passive Replicates in Delta-4 to provide Dependable Distributed Computing, In Proceedings of the 19th International Symposium on Fault-Tolerant Computing pp. 184-190, 21-23 June 1989 Chicago, Illinois.

[SSh90] Santosh K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, D. T. Seaton, Fail-Controlled Computer Architectures for Distributed Systems, Computing Laboratory, University of Newcastle upon Tyne, NE1 7RU, UK.

[SY85] Strom, S. Yemini, Optimistic Recovery in Distributed Systems. ACM Transactions on Computer Systems, 3(3), pp.204-226, August 1985.

[Th89] Thambidurai, A. M. Finn, R. M. Kieckhafer, J. C. Walter, Clock Synchronization in MAFT, In Proceedings of the 19th International Symposium on Fault-Tolerant Computing pp. 142-149, 21-23 June 1989 Chicago, Illinois.

[TNG] E. Ruohtula, P. Hjort, M. Lindqvist TNSDL Grammar version 1.8. Nokia Telecommunications Jan. 1991.

[TNS91] M. Lindqvist, E. Ruohtula, E. Kettunen, H. Tuominen The TNSDL Book, Third Draft Edition Telenokia, March 1991.

[To78] Toy, Fault Tolerant Design of Local ESS Processors, in Proceedings of the IEEE, vol. 66 No.10 Oct.1978 1126...1145. Also in Victor P. Nelson and Bill D. Carroll Tutorial: Fault Tolerant Computing IEEE Computer Society press.

[Tr89] Tripathi, An Overview of the Nexus Distributed Operating System Design IEEE Transactions on Software Engineering. Vol 15, No 6, June 1989.

[We78] Wensley, L.Lamport, etc. SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control, in Proceedings of the IEEE Vol.66 No.10, Oct.1978, 1240...1255. Also in Victor P. Nelson and Bill D. Carroll Tutorial: Fault Tolerant Computing, IEEE Computer Society press.

[Ya89] Takahiko Yamada, Satoshi Ogawa, Fault Tolerant Multiprocessor for Digital Switching Systems. In Proceedings of the 19th International Symposium on Fault-Tolerant Computing pp. 245-252, 21-23 June 1989 Chicago, Illinois.

[Yi80] Ying W.NG, Algirdas A. Avizienis, A Unified Reliability Model for Fault-Tolerant Computers IEEE Transactions on Computers, Vol. C-29, No 11, Nov 1980 1002...1011. Also in Victor P. Nelson and Bill D. Carroll Tutorial: Fault Tolerant Computing IEEE Computer Society press.

[zB90] Özapl Babaoglu, Fault-Tolerant Computing Based on Mach ACM Special Interest Group on Operating Systems SIGOPS Vol. 24 No.1 Jan 1990 p.27...39.

[Z.100] CCITT Specification and Description Language SDL. CCITT Recommendation Z.100, Geneva, March 1988.

Appendix 1: Resynchronisation

Notation: R_{wo} is the hand process in the active unit,
R_{sp} is the corresponding hand process in the spare unit,
Inque is the incoming message queue of a process,
By freeze operation a process is put into a state in which it will not be given any CPU time and thus all new incoming messages will be queued.

Active unit

Warm-up control:

```
state=idle
DO FOREVER
   main-receive
   if state=idle and RESYN_REQ
      freeze $R_{wo}$
      if RESYNC ALLOWED
         send RESYN_SP
         set   rT
         state=wfS
      else
         send RESYN_ACK(wo+sp,NOK)
         defreeze $R_{wo}$
      fi
   elseif state=wfS and SLEEP_ACK
      if INQUE($R_{wo}$) = empty
         send DATA->SP
         if not LAST
            state=wfA
         else
            state=wfE
         fi
      else
         send RESYN_ACK(wo+sp,NOK)
         defreeze $R_{wo}$
         send NACK_END
         reset rT
         state=idle
      fi
   elseif state=wfS and NACK_SLEEP
      send RESYN_ACK(wo+sp,NOK)
      defreeze $R_{wo}$
```

Spare unit

Warm-up control:

```
state=idle
DO FOREVER
   main-receive
   if state=idle and RESYN_SP
      freeze $R_{sp}$
      if $R_{sp}$ in main-receive
         $Lr_{sp}$ = $R_{sp\_}$inque_length
      send SLEEP_ACK
         set rrT
         state=wfD
      else
         send NACK_SLEEP
         defreeze $R_{sp}$
      fi
   elseif state=wfD and DATA
      if $Lr_{sp}$ > 0
         delete first $Lr_{sp}$
         messages in $R_{sp}$ inque
         $Lr_{sp}$ = 0
      fi
      write DATA
      if not LAST
         send ACK_OK
      else
         send END_OK
         set "Replication mode ok"
         defreeze $R_{sp}$
         reset rrT
         state=idle
      fi
   elseif state=wfD and (NACK_END
        reset   rT
        state=idle
    elseif state=wfA and ACK_OK
        send DATA->SP
        if LAST
            state=wfE
        fi
    elseif state=wfE and END_OK
        set "Replication_mode_ok"
        send RESYN_ACK(wo+sp,OK)
        defreeze $R_{wo}$
        reset rT
        state=idle
    elseif state= not idle and  rT=0
        send NACK_END
        send RESYN_ACK(wo+sp,NOK)
        defreeze $R_{wo}$
        reset rT
        state=idle
    fi
ENDDO
```

```
                    or rrT=0)
        defreeze $R_{sp}$
    state=idle
    fi
ENDDO
```

Appendix 2: Safe Synchronisation

Notation: R_{wo} is the (hand) process executing the safe transition in the active unit,
R_{sp} is the corresponding spare process.

Active unit

```
Begin_Safe_Trans(safety prop.msg)
    if SP exists
        select free SYNC SERVER
        send SAFSYN_REQ to WO SYNC SERVER
        receive-with-timeout SAFSYN_ACK
        if SAFE SYNC was OK
            set "Replication mode ok"
            cascade time-limits
        fi
    fi
    return
END

SYNC  SERVER
Notify: I am free
state = idle
DO FOREVER
    main-receive
    if state=idle and SAFSYN_REQ
        if SYNC- ALLOWED
            count WO_CHSUM
            send SAFSYN_SP
            set rT
            state=wfA
        else
            send SAFSYN_ACK(wo+sp,NOK)
            Notify: I am free
        fi
    elseif state=wfA and ACK_OK
        if INQUE($R_{wo}$) has not changed
            DO while not LAST and less than N
                send DATA ->SP
                count messages
                state=wfE
            OD
        else
            send SAFSYN_ACK(wo+sp,NOK)
            defreeze $R_{wo}$
            send NACK_END_TO_SP
```

Spare unit

```
Begin_Safe_Trans(safety prop. msg)
    send SAFSYN_IND to SP
            SYNC SERVER
    receive-with-timeout SAFSYN_ACK
    if SAFE SYNC was OK
        set "Replication mode ok"
        cascade time-limits
    fi
    return
END

SYNC  SERVER:
state = idle
DO FOREVER
    main-receive
    if state= idle and SAFSYN_SP
        if safety prop.msg has NOT arrived
                to Rsp
            write safety prop. msg
            to $R_{sp}$ inque last from SAFSYN_SP
        fi
        set ssT
        state=wfRsp
    elseif state=idle and SAFSYN_IND
        set ssT
        state=wfWo
    elseif (state=wfWo and SAFSYN_SP)
        or(state=wfRsp and SAFSYN_IND)
        reset ssT
        count SP_CHSUM
        if WO_CHSUM=SP_CHSUM
            send SAFSYN_ACK(wo+sp,OK)
            state = idle
        else
            write DATA from SAFSYN_SP
            if LAST
                send SAFSYN_ACK(wo+sp,OK)
                state = idle
            reset rT
            Notify: I am free
            state  idle
        fi
    elseif (state=wfA or wfE) and rT=0
        Notify: I am free
        state=idle
    fi
ENDDO
```

```
            else
                set rrT
                state = wfD
                send ACK_OK
            fi
        fi
    elseif state=wfD and DATA
        write DATA
        if LAST and all synchoronised
            reset rrT
            send SAFSYN_ACK(wo+sp,OK)
            state=idle
        fi
    elseif (state=(wfRsp or wfWo) and ssT=0)
        or (state=wfD and rrT=0)
        send SAFSYN_ACK(wo+sp,NOK)
        state=idle
    elseif  state=wfD and NACK_END_TO_SP
        reset rrT
        state=idle
    fi
ENDDO
```