High available and fault tolerant mobile communications infrastructure

Beiroumi, Mohammad Zib; Iversen, Villy Bæk; Dittmann, Lars

Publication date:
2006

Document Version
Publisher's PDF, also known as Version of record

HIGHLY AVAILABLE AND FAULT TOLERANT MOBILE COMMUNICATIONS INFRASTRUCTURE

By

Mohammad Zib Beiroumi

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Technical University of Denmark
(Research Center COM)
2005

ABSTRACT

High availability is a key requirement in mobile communication systems, especially, when it is used for mission-critical services such as public safety e.g. police, ambulance and fire services. A failure in the fixed network infrastructure that provides services to mobile users can affect a large number of users and risk loss of lives. The fixed infrastructure of mobile communication system has different characteristics, for example, architecture complexity, real-time peer-to-peer communication and performance requirements that make the already existing failure recovery techniques, such as those using rollback or replication techniques inapplicable.

This dissertation presents a novel failure recovery approach based on a behavioral model of the communication protocols. The new recovery method is able to deal with software and hardware faults and is particularly suitable for mobile communications infrastructure. The method enables the faulty applications in the infrastructure to quickly and effectively resume their services to their mobile clients with no or minimal loss of work after failure.

In our approach, we do not assume a specific fault behavior for example fail-stop or transient behavior as it is the case for many recovery techniques. In addition, the method does not require any modification to mobile clients. The Communicating Extended Finite State Machine (CEFSM) is used to model the behavior of the infrastructure applications. The model-based recovery scheme is integrated in the application and uses the client/server model to save the application state information during failure-free execution on a stable storage and retrieve them when needed during recovery. When and what information to be saved/retrieved is determined by the behavioral model of the application.

To practically evaluate and demonstrate the effectiveness of our method, we developed as a case study an experimental testbed for the TETRA (TErrestrial Trunked Radio) packet data network. The testbed works as a distributed system and can run various communication scenarios between the fixed network infrastructure and its mobile users. We thoroughly followed the TETRA standard specifications in our implementation of the communication protocols in order to get a testbed system that operates as the real system with respect to message exchange and timing. The experimental results showed that by using our method the faulty infrastructure application can immediately resume its service after its restart and in less than a minute, it restores its service performance level prior to the failure. The failure-free overhead incurred by the method is relatively low, and is experimentally found to be less than 5% in the conducted experiments.

RESUME

Høj tilgængelighed er et nøgle-krav til mobilkommunikationssystemer, i særdeleshed når de anvendes til livsvigtige opgaver som offentlig sikkerhed, f.eks. politi, ambulance- og brandtjeneste. En fejl i infrastrukturen af det faste netværk, som tilbyder mobile brugere tjenester, kan påvirke et stort antal brugere med risiko for tab af menneskeliv. I sammenligning med det faste netværk har infrastrukturen i mobilkommunikationssystemer helt andre karakteristika, så som kompleksiteten af arkitekturen, realtid peer-to-peer trafik og krav om ydelser. Dette gør, at man ikke kan anvende eksisterende genetablerings teknikker som tilbage-rulning eller Duplikering.

Denne afhandling præsenterer en ny genetablerings metode, der er baseret på kommunikations protokollernes adfærd. Denne nye metode er i stand til at tolerere fejl i både programmél og maskinél og er i særdeleshed egnet til de forhold, der hersker for mobilkommunikation. Metoden gør det muligt hurtigt og effektivt at genetablere de af fejl i infrastrukturen ramte tjenester til de mobile kunder men ingen eller et minimalt tab af arbejde. I modsætning til mange andre genetableringsteknikker stiller vores metode ikke specifikke krav til den måde, fejl opfører sig på, så som ophør af fejl eller forbigående fejl. Endvidere kræver metoden ingen modifikation af de mobile enheder. Den kommunikerende-udvidede-begrænsede-tilstands-maskine (Communicating Extended Finite State Machine, CEFSM) anvendes til at modellere opførslen af

de infrastruktur programmerer som servicerer mobiler. Den model-baserede genetablerings metode er integreret i selv programmet. Den bruger klient/server modellen til at gemme oplysninger om programmets tilstand i fejlfrie perioder og hente dem frem igen, når de skal bruges til genetablering. Oplysninger gemmes i en pålidelig opbevaring plads. Hvilke oplysninger der skal gemmes, og hvornår de skal gemmes/hentes, fastlægges ved hjælp af modellen for programmets tjeneste adfærd.

For at vurdere og underbygge vores metodes effektivitet i praksis, har vi som eksempel opbygget en eksperimentel forsøgsmodel af TETRA (TErrestrial Trunked Radio) pakkekoblede datanet. Forsøgsmodellen fungerer som et distribueret system og kan afvikle forskellige scenarier for kommunikationen mellem det faste nets infrastruktur og dets mobile brugere. For at få en forsøgsmodel, der fungerer som det virkelige system med hensyn til informationsudveksling og tidsmæssigt forløb, fulgte vi ved implementeringen af kommunikations protokollerne omhyggeligt specifikationerne for TETRA standarden. De eksperimentelle resultater viser, at med vores metode kan en i infrastrukturen fejlramt tjeneste efter genstart reetableres til samme niveau som før fejlen indtraf i løbet af få sekunder. Den ekstra belastning, som metoden medfører i fejlfrie perioder, er ret lille, og i alle eksperimenter har den været mindre end fem procent.

AUTHOR PUBLICATIONS

Part of the research presented in this thesis has also been published in the following papers (in chronological order):

- M. Zib Beiroumi, High Available Mobile Infrastructure Applications, proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005), pp. 181-190, Chicago, USA, Nov, 2005.
- M. Zib Beiroumi, V. Iversen, Recovery method based on communicating extended finite state machine (CEFSM) for mobile communications, proceedings of the 10th IEEE International Conference of Engineering of Complex Computer Systems, pp. 384-393, Shanghai, China, June, 2005.
- M. Zib Beiroumi, Recovery of Infrastructure Software in the Mobile Network, NTS-17, 17th Nordic Teletraffic Seminar, pp. 137-148, August 25, 2004, Fornebu, Norway.
- M. Zib Beiroumi, Recovery of peer-to-peer applications in the mobile network infrastructure, Fast abstract in the IEEE International Conference on Dependable Systems & Networks (DSN-2004), pp 62-63, June 28-July 1, 2004, Florence, Italy.

ACKNOWLEDGMENTS

I would like first to thank my manager Lars Behrendt at Motorola who has supported me from the beginning to the end of this joint PhD research project between Motorola and the Technical University of Denmark. I sincerely doubt that this project would have ever seen the light without his help and support to overcome all the obstacles that stood in the way.

I would also like to thank my advisors Villy Bæk Iversen and Lars Dittman for their great support and contribution to the success of this project.

Last but not the least I must thank my family. I am fortunate to have such a wonderful wife, who has supported me all the way through and has lived up to the challenges at home with our three very active kids.

My research work at the Technical University of Denmark was totally funded by Motorla A/S Tetra world, sydvestvej 15, 2600 Glostrup, Denmark.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

BS	Base Station
CEFSM	Communicating Extended Finite State Machine
FNI	Fixed Network Infrastructure
FPS	File Packet Sender
GPRS	General Packet Data Service
IP	Internet Protocol
LLC	Logical Link Control
MAC	Medium Access Control
MEU	Mobile End User
MLE	Mobile Link Entity
MS	Mobile Station
MSC	Message Sequence Charts
MTBF	Mean Time Between Failure
MTTR	Mean Time To Repair
OSI	Open Systems Inter-connection
PDS	Packet Data Server
PDU	Packet Data Unit
SAP	Service Access Point
SDL	Specification and Description Language
SDU	Service Data Unit
SIS	State Information Saver
SNDCP	SubNetwork Dependent Convergence Protocol
STBR	State Transition Based Recovery
TETRA	TErrestrial Trunked Radio
UML	Universal Modeling Language
UMTS	Universal Mobile Telecommunications System

Chapter 1
Introduction

In building public safety communication systems that aim to tolerate failure, system developers must tackle many difficult issues. For example, there is the issue of which failure recovery approach that can work best for the system? What type of faults should the system tolerate? Does the system performance or real-time requirements deteriorate during failure-free execution? What is the cost of adding fault tolerance to the system? By systemically building a method to tolerate failures caused by software and hardware faults, we endeavor with this study to illuminate many of these issues.

1.1 Motivation

Mobile communication is a key element for the success of the public safety work and it is a necessary tool in solving the day-to-day mission-critical tasks accomplished by public safety services such as ambulance services, fire brigades and police forces. The public safety workers are expected to provide prompt assistance in dealing with situations to preserve life, health and security. It is therefore very important that the public safety services have reliable and highly available mobile communication infrastructure in place to support the needs of the public.

In the last four decades, researchers have developed different techniques to tolerate system failures. There are three main approaches used to develop these techniques:

1. *Rollback*: In this approach, the state of the application is saved periodically during failure-free execution to a stable storage. In case of failure, the faulty application is rolled back to the latest saved state and tries to recover from there.
2. *Replication*: In this approach, the application (mainly server) is replicated and distributed across different computers. The idea behind this approach is that the failure of one server replica (or of a computer hosting a replica) can be masked from any client using that server because the other replicas can continue to perform any operation that the client requires from the faulty server.
3. *Design diversity*: This approach is based on the use of two or more versions of the application that are built independently (i.e. different designers, different programming languages, different development tools, etc.) from the same specifications. The rational for this approach is that the different versions fail independently because it is unlikely to have faults at exactly the same place in all versions, and thus, the probability of having at least one running application at any time is very high.

Unfortunately, these approaches suffer from different limitations that restrict their use in commercial communication systems, for example, because of implementation cost or some inadequate assumptions about the causes to failures. In addition, mobile communication systems have many specific requirements such as real-time and performance requirements that seriously challenge the applicability of these approaches in mobile environment. In communication industry today, most of the system

suppliers have their own customized solutions and approaches to deal with their systems failures. These solutions and approaches are based on best practice rather than on scientific studies.

Our goal with this thesis is to develop a failure recovery method that gives a realistic solution for achieving fault tolerance in real-world communications systems. We are particularly interested in mission-critical public safety communication systems because of the obvious need for continuous service availability. The proposed recovery method should improve the system availability through fast and reliable recovery. The method should also meet the requirements of today's enterprise such as low implementation cost, good scalability, low overhead during failure-free execution, etc.

1.2 Scope and contributions

This dissertation details our research work to develop a failure recovery method to achieve high availability in mobile communication infrastructure. We investigate the challenges that in mobile environment create for the recovery and try to develop a scientific and engineering quality solution.

The path we take in our work cuts a broad swath through traditional systems and fault tolerance research. We look at the existing recovery approaches and explain their general limitations. We then describe the characteristics of mobile communication and their impact on the recovery approach. We begin by constructing a model that formally describes the behavior of the mobile communication protocols. The behavioral model is then used in our development of the failure recovery method. Finally, we implement an experimental testbed for real-world TETRA packet data communication system to evaluate our proposed method. Our work led to the following contributions:

1. Adaptation of the CEFSM model to the OSI model in order to get a more accurate behavioral model of the communication protocols which are normally designed according to the OSI model.
2. Studying fault tolerance in a new application area, namely the fixed infrastructure of mobile communication system. The fixed infrastructure manages and provides services to the mobile stations. To the best of our knowledge there is no academic literature that deals with failure recovery in the fixed infrastructure but there is few for mobile stations, e.g. [Pradhan96].
3. Applying the CEFSM model on a real-world case study, namely TETRA packet data communication system. We show how behavioral model can be developed for the layered communication protocol stack.
4. Developing a novel behavioral model based failure recovery method to tolerate software and hardware faults. This recovery method referred to as State Transition Based Recovery (STBR) is aimed to achieve high availability in the mobile fixed network infrastructure. The method is well suited for real-time communication and do not rely on any specific fault behavior e.g. transient or fail-stop.
5. Design and implementation of an experimental testbed for TETRA packet data where the communication between mobile users and infrastructure can be generated at various traffic profiles.

1.3 Dissertation Overview

In this dissertation, we gradually assemble the pieces needed to first develop our novel failure recovery method for mobile communication systems, and secondly to implement an experimental testbed to evaluate the proposed method.

In chapter 2, we present the most known recovery techniques in the field of fault tolerance research. We describe in general the basic ideas and approaches behind these techniques. Finally, we explain their general limitations as a result of the assumptions made by these approaches.

In chapter 3, we look at the overall architecture of mobile communication systems and the different aspects that characterize mobile communication and its physical environment. We then investigate the implications of these characteristics on the existing recovery techniques and finally come to a number of requirements that should be considered when building recovery system for mobile environment.

In chapter 4, we start by presenting a modified version of the CEFSM model that is adapted to the OSI model. We then introduce our case study about TETRA and describe the protocol stack of TETRA packet data. Finally, the CEFSM model is applied to a selected layer protocol entity in the TETRA packet data protocol stack.

In chapter 5, we present our failure recovery method. The principles that the method relies on, the objective and the assumptions are described in details. Furthermore, the protocol entity modeled in previous chapter is used to explain the recovery mechanism both during failure free execution and during recovery.

In chapter 6, the design and implementation of the experimental testbed for TETRA packet data is described. The experiment procedure and setup is also described. Finally, the results of the conducted experiments are presented and evaluated.

In chapter 7, we conclude our work and discuss the possibility of adopting our research in commercial systems.

1.4 Terminology

The fundamental terms used in the field of fault-tolerance research are fault, error, failure, detection and recovery. The terminology used in this thesis is to some extent in line with that given by [Gray91] and [Pradhan95].

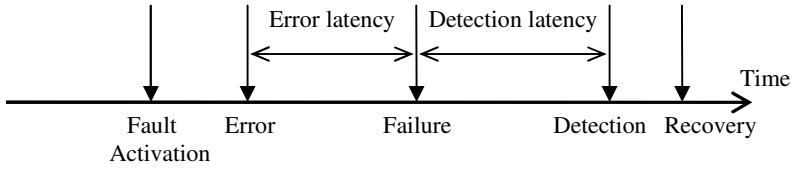

Figure 1 A generic time line from fault to recovery

A *fault* is a physical defect that may lead to an error. Faults can be classified into different types such as:

Hardware faults: component failures, for example disk crashes or processor failures

Software faults: faults in software e.g. coding mistakes or improper design

Human faults: mistakes made by operators and maintenance personnel, for example making an erroneous change to a configuration file, or performing a failed upgrade.

Environmental faults: failure in facilities e.g. fire, flood, earthquake, power failure and sabotage.

In case of software, faults are activated or triggered when the faulty piece of code is executed. The *fault activation rate* measures how often faults are triggered.

An *error* is an erroneous change in the system state caused by the activation of the fault. It is a deviation from the correct behavior of a system. Fault is the root cause of error and error may cause failure.

A *failure* is the nonperformance or incorrect performance of some action that is expected of the system by the user.
Error latency is the time between error occurrence and the failure occurrence, see Figure 1.
Failure detection is the process of identifying that the system is in failed state. There are different failure detection mechanisms that can work both locally (e.g. by using watchdog timers) and remotely (e.g. by using periodic heartbeat messages) to monitor the system state.
Failure detection latency is the time between the failure occurrence and its detection by the deployed detection mechanism.
Failure recovery is the process of getting the system back to an operational state after a failure has been detected.

To illustrate the above mentioned terminology, a simple C code example containing out-of-bounds array indexing fault is presented in Figure 2. The fault is located at line code number 5 (≤ should be replaced by<), the fault is activated when line 9 is executed with parameter *i* equals to 100 which takes place when the user list is full. The fault activation will lead to writing the number out of the array bounds, precisely at address "`&user_list[100]`".
The fault activation will cause an error if the memory address "`&user_list[100]`" is already used by another variable in the program, for example if the compiler uses this address for the variable `number_of_users`, otherwise no error occurs. Suppose that an error is indeed occurred (i.e. `number_of_users` gets corrupted) then depending on the program flow, if the `number_of_users` is used before being overwritten then program failure is inevitable, but if it is always overwritten before being used then failure is avoided. The possible failure will occur at line 22 where the program depending on the error value may hang for a variable period of time or probably crash.

```
#define MAX_NUM_OF_USERS              100
int  user_list[MAX_NUM_OF_USERS];
int  number_of_users


1 Save_user_id_number(int id_number)
2 {
3   int i;
4   /* save Id number in first empty element */
5    for ( i= 0; i ≤ MAX_NUM_OF_USERS;i++)
6    {
7      if(user_list[i] == 0)
8      {
9        user_list[i] = id_number;

           ⋮

  }
20 send_to_all_users( )
21{
22      while(number_of_user--)
        {
           ⋮
```

Figure 2: An example of out-of-bounds array indexing in C code

Error latency is the time period between the execution of line 9 with parameter *i* equals to 100 and the following execution of line 22. Note that error latency is a variable that may depend on the program user activity. The failure detection latency is the time period between the execution of line 22 and the detection of failure by the used detection techniques, e.g. via heartbeats and watchdog timers.

Finally, there are two important properties for software faults that have been essential for many of the failure recovery methods [Chandra00a].

Non-Determinism: This property indicates that fault activation is non-deterministic (transient) and it is most likely not to happen if the operation is retried, even if the same piece of code is retried. The transient nature of the

fault arises because some external factors have unexpectedly changed; for example, a race condition caused by unusual thread/process scheduling or a bit-flip (a change from 0 to 1 or 1 to 0) in RAM caused by electromagnetic interference. The non-deterministic software faults are also known as "Heisenbugs" [Gray86]. The faults that do not uphold the non-deterministic property are known as permanent faults.

Fail-stop property: A program must not perform erroneous actions after fault activation, for example writing erroneous data which corrupts its own process state or sending incorrect information to other processes. This property is also known as halt-on-failure. The faults that do not uphold fail-stop property is called Byzantine faults [Schneider84].

Chapter 2

Fault Tolerance: Recovery techniques & limitations

Traditionally, fault tolerance means to avoid service failures in the presence of faults. The goal of fault tolerance is to mask or at least to minimize the impact of system failures on system users. Fault tolerance is a means to achieve high level of system availability. In this chapter, we describe the key principles to build fault tolerant system and the main existing techniques to achieve fault tolerance. Finally, the limitations of these techniques are explained.

2.1 Availability and Reliability

Reliability and availability are two metrics that are always related with fault tolerance.

Reliability is the probability that a system will not fail at a specified point of time in the future given that it is operating correctly at time zero. Module reliability measures the time from an initial instant and the next failure event. Mean Time Between Failure (MTBF) is used to statistically quantified reliability. MTBF is the mean (average) time expected between failures of a given module (software or hardware) and is normally measured in hours. Because the calculation of MTBF is quite complex and may depend on many factors, it is usually done empirically to predict the rate at which failures can be expected.

In contrast, availability is the probability that the system will be operating correctly at any instant of time within a given time interval. A widely accepted equation for system availability is A = MTBF/(MTBF+MTTR), where MTTR (

Mean Time To Repair) is the average time between failure and recovery. An ideal system that never fails has availability equals to 1. Availability measures the readiness for correct service, while reliability measures the continuity of correct service.

Depending on the criticality level of the user application, the requirements for system reliability and availability may vary. For example, in mission-critical applications such as emergency services, the main concern is a high level of availability; few numbers of outages per year can be tolerated as long as they are very short. While for life-critical application such as control system for nuclear power plants, high reliability is the main concern since no failure can be tolerated during the life of the system. Reliability and availability are related in such a way that improving module reliability will automatically improve its availability, but the reverse is not necessary true.

2.2 Software Faults

A system can be viewed as a set of modules - hardware and software – that communicate with each other through network (wired or wireless) to achieve common goals. Each module is designed to perform a specified number of functions and it has a well defined interface through which it can interact with other modules. A module may also divided into several sub-modules if it is large.

Systems fail due to a variety of problems with their software and hardware. Field studies [Gray91] and everyday experience show that the dominant cause of failures today is software faults, both in the application and operating system. We mean by "application" any software module that runs over the operating system ranging from end-user applications to system applications.

As previously mentioned, software fault is the root cause of error and possible consequent failure. It is a defect that is located in a fixed position in a

specific module. However, the error that is caused by the software fault may not be limited to a single module and it may propagate to other modules. Consider the case where the faulty module starts sending corrupted messages to other modules, if the receiving modules are not prepared to handle such errors, they may fail too.

The behavior of the fault is critical to the success of any proposed recovery method. Recall that recovery can be first started after detection of the failure so the evolvement from fault activation to failure detection is important to understand in order to ensure a successful recovery procedure after a failure. Most of the existing failure recovery techniques have some assumptions to the behavior of the software faults that they can tolerate.

2.3 Building fault tolerant systems

To build a fault tolerant system, there are four key elements to be addressed. A lack of any one of these elements will make the system less fault tolerant.

1. *Redundancy*: A fault tolerant system must not have any single point of failure, therefore, both hardware and data redundancy is necessary to recover from hardware and software faults. The principle of redundancy relies on the fact that the probability of two or more redundant components failing at the same time is very low assuming that there is no dependency between them.
2. *Modularity*: A fault tolerant system should be decomposed into modules where each module (software or hardware) is a unit of service with a well defined access interface. Besides that modularity is an important design approach to break down the complexity of the system, it is also an affective approach to hinder error propagation by adding strict error control at every access point to each module.

3. *Failure detection*: The main goal of failure detection is to determine when the system (most probably) does not operate correctly and to give the start signal for recovery/repair procedure. The quality of failure detection can be evaluated by two metrics: detection promptness which is directly translated to failure detection latency and second detection reliability which is the probability that the failure decision is wrong (false alarm). There are different failure detection techniques that can be used separately or in combination. Examples of failure detection techniques are: watchdog timers to detect hanging processes and heartbeat messages to detect crashed processes.
4. *Failure recovery*: A fault tolerant system should be able to resume service after a failure and to bring the system state to that it had before failure. The aim of failure recovery is to reduce users' loss of work as well as to minimize redo. A fundamental task for failure recovery process is to bring consistency to the system after failure. Failure recovery may need to utilize any of the above mentioned elements e.g. redundancy to achieve its goal. A vast number of failure recovery techniques have been proposed in the literature to achieve fault tolerance in the distributed systems. We treat some of the most known techniques in the next section.

2.4 Failure Recovery Techniques

During the last four decades, different failure recovery techniques and approaches have been developed. We describe some of the most known recovery methods in the field of fault-tolerance research.

2.4.1 Rollback recovery

Rollback recovery regards the system as a collection of application processes that communicate through network. The rollback recovery protocols

try to achieve fault tolerance by saving the complete process state of the application periodically on stable storage during failure-free execution. Upon a failure, the failed process rolls back to its latest saved state and then tries to recover from there, thereby reducing the amount of lost computation. The main goal of any rollback recovery protocol is to bring the system (processes) into a consistent state when inconsistencies occur because of a failure. Rollback recovery techniques try to achieve transparent recovery by avoiding the need of any involvement of the application programmer and just treat the application to be recovered as a black box. The Rollback recovery can be classified into two groups [Elnozahy96]: checkpoint based and log-based.

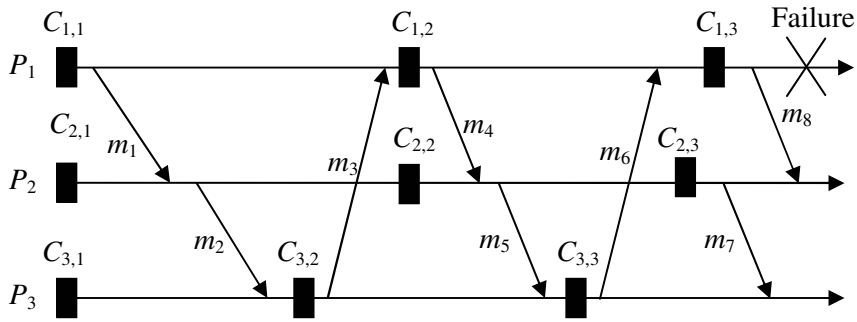

Figure 3: Rollback propagation and domino effect

- **Checkpoint based Recovery**: checkpoint-based rollback recovery relies on checkpoints to achieve fault tolerance. A check point is a "snapshot" of the process state at a certain point of time as maintained by the operating system (program counter, data segments, CPU registers, stack pointers, etc) . Upon a failure, checkpoint-based rollback recovery restores the system state to the most recent consistent set of checkpoints. The simplest form of checkpoint based schemes referred to as *uncoordinated* where each process can conveniently take checkpoints according to some local criteria, for example to reduce performance

overhead, without taking account to the communication messages with the rest of the system. Uncoordinated checkpointing is simple to implement but it suffers from *domino effect* [Randell75] which may cause loss of work. After failure, the failed process rolls back to the latest saved checkpoint but this may not be in consistency with the latest checkpoint of one of the other processes, so that process is obliged to roll back to next older checkpoint. This cascaded rollback may continue and eventually may lead to the domino effect, which causes all processes of the system to roll back to the beginning of the computation, in spite of all the saved checkpoints. Consider the example in Figure 3, the system in this case is composed of three processes P_1, P_2 and P_2. Each process takes a checkpoint – represented by black bar independently. Suppose process P_1 fails and rolls back to checkpoint $C_{1,3}$. The rollback of P_1 invalidates the sending of message m_8 and so P_2 must rollback to checkpoint $C_{2,3}$ to "invalidate" the receipt of that message. Consequently, the rollback of P_2 will force the rollback of P_3 to check point $C_{3,3}$ to invalidate the receipt of message m_7. This cascaded rollback continues until all processes roll back to their initial checkpoints ($C_{1,1}$, $C_{2,1}$, $C_{3,1}$). To avoid domino effect, c*oordinated* checkpointing where processes coordinate their checkpoints in order to save a consistent global system state [Chandy85] is used. A coordinator process takes a checkpoint and broadcasts a request message to all other processes, requesting them to take checkpoint. Checkpoint coordination can also be achieved by using synchronized clocks where all system processes take checkpoints at approximately the same time without need to a coordinator process [Cristian91].

- **Log-based checkpointing**: Log-based rollback-recovery uses both checkpointing and logging to enable processes to replay their execution after a failure beyond the latest checkpoint. Log-based checkpointing is

useful for the systems that interact frequently with the outside world. The systems in the outside world can not be rolled back and thus instead of taking expensive checkpoints whenever messages received/sent from/to outside world, it is enough to log messages and replay them after failure. Log-based recovery relies on the *piecewise deterministic* assumption [Strom and Yemini 1985]. This assumption assumes that the rollback-recovery protocol can identify all the nondeterministic events (e.g. receiving messages from the outside world and asynchronous interrupts) executed by each process, and for each such event, logs a determinant that contains all information necessary to replay the event should it be necessary during recovery. There are different flavors of log-based recovery depending on how the determinants are logged to a stable storage, *pessimistic* and *optimistic* are two best known log-based recovery techniques. Pessimistic log-based assumes that a failure can occur after any nondeterministic event in the computation and therefore the process is blocked after each nondeterministic event waiting for its determinant to be logged to a stable storage before processing the event. Pessimistic logging simplifies the recovery and rolls back to a system consistent state that is very close to the pre-failure state, but the cost to pay is a high failure-free performance overhead. In contrast, Optimistic logging [Storm85] assumes that determinants will be logged to stable storage before a failure occurs because failures are normally infrequent and thus there is no need to interrupt the process on every nondeterministic event. Determinants are kept on volatile log that is periodically flushed to stable storage. Optimistic log-based recovery achieves low failure-free overhead but uses a rather complex recovery scheme.

Rollback based Recovery has focused traditionally on recovering long running scientific computations [Casanova97], text editors [Lowell00], spreadsheet programs and database systems [Campos95].

2.4.2 Replication based recovery

Replication implements roll-forward mechanism where the entity (mainly a server application) is replicated to establish a group of replicas and in the event of the failure of one entity, the other replicas can take over and continue processing requests. There are two best-known replication approaches:

- *Active replication*: In active replication [Schneider93] (also known as state-machine approach), all server replicas run concurrently and execute the same work so they maintain exactly the same consistent state. Every server replica processes every client request in the same relative order and sends back a reply. Figure 4 illustrates schematically the architecture of active replication with three server replicas. Reliable multicast protocols may be used to forward client requests to all members of the server group. Majority voting technique is used when the group consists of more than two members to deliver the correct reply to the client. If the fail-stop property is assumed then in order to tolerate k number of faulty replicas , a group of k+1 replicas is enough because faulty replicas keep silent and do not send any incorrect replies. However, if the Byzantine property is assumed then 2k+1 replicas is needed to sort out a possible k incorrect replies from the k faulty replicas. Active replication is very effective for hardware faults and provides a fast recovery. Active replication can also be used for load balancing by equally distributing clients' requests on all members of the server group.

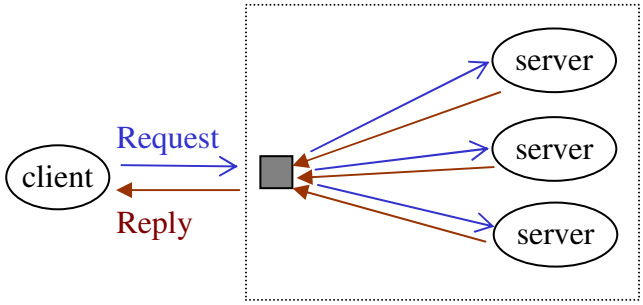

Figure 4: Active replication structure

- *Passive replication*: In passive replication (also known as primary-backup) [Budhiraja93], one member of the server group is designated as the primary, while all other replicas serve as backups. The primary server is the only one that process clients' requests and send back replies. During normal operation, the state of the primary is periodically recorded in a log, typically as a sequence of request and reply messages, while?? states and updates as checkpoints. Upon a failure, a backup server is promoted to be the new primary server of the group. The state of the new primary is restored to the state of the old primary by reloading its state from the log, followed by reapplying request messages recorded in the log.

The replication techniques have been mainly used in building enterprise distributed applications such as databases and transaction processing systems. Some of the best known systems that used replication to achieve fault tolerance in the enterprise applications are SIFT [Wensley72], ISIS [Birman94], and AQUA [Cukier98].

2.4.3 N-version programming

N-version programming [Avizienis77] uses design diversity approach and it is defined as the *independent generation* of $N \geq 2$ functionally equivalent programs from the same *initial specification*. Independent generation of programs means that the programming efforts are carried out by *N* development teams that do not interact with respect to the programming process. The initial specification is a formal specification in a specification language. The goal of the initial specification is to state the functional requirements completely and unambiguously, while leaving the choice of implementations to the N programming efforts. N-version programming assumes that all programs contain faults, but it relies on the fact that the number of hidden faults will be small and that they will be in different locations in each of the versions. Wherever possible, different algorithms, programming languages and compilers are used in each separate effort.

Figure 5 shows the basic structure of the N-version programming scheme. The N programs run concurrently and the results of each version compared and voted on to determine the final output.

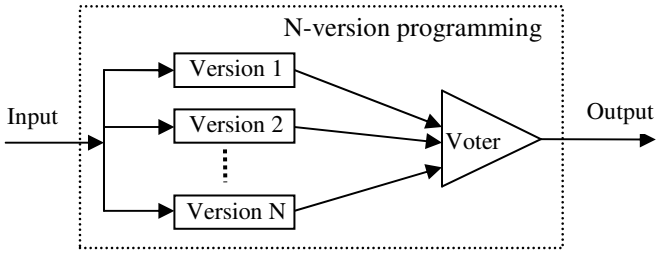

Figure 5: N-version programming structure

The case where N equals two is a special case since no majority outcome can be derived when the individual program results do not agree. Most fault-tolerant software systems utilize $N \geq 3$, and use a majority vote to resolve inconsistent results. Note that N-version programming is not only a recovery technique but it also provides failure detection through voting mechanism.

The N-version programming is mainly utilized in life-critical applications with a high risk of life loss for example flight control computers e.g. in Boeing 737-300 [Williams83] and Airbus 320 [Traverse88].

2.5 Limitations

In this section, we explain the general limitations of these techniques and leave the specific ones concerning the mobile environment to the next chapter.

- Rollback depends on two assumptions:
  1- Transient faults: Without assuming that faults are transient, the faulty process will certainly fail again at exactly the same place. The faulty process will roll back to the latest saved state and then continues its execution (exactly the same program instructions are repeated) to restore the pre-failure state before it hits the error again. Note that the faulty entity may or may not reactivate the permanent fault depending on the latest checkpoint time, but it will certainly hit the error.
  2- Good checkpoints: Rollback assumes that only good data is saved to a stable storage and this implies that the fail-stop property must be upheld. In other words, the saved states must not contain the error that is caused by the transient fault.

- Replication has also two assumptions

1. Transient fault: Replication approach depends on the assumption that most of the software faults are transient. If this assumption is not applied, then all members of the replica group will fail at the same time, for example because of a permanent software bug.
2. Fail-stop: Most of the replication techniques assume fail-stop property, i.e. an entity works correctly or stops functioning completely. This assumption can be relaxed at the cost of more complex voting algorithm and an increase in the number of replicas.

- N-version or the use of diversity has no technical limitation in general, but its main limitation is its high cost both with respect to implementation and maintenance. There is a big discussion whether it is better to concentrate on developing one reliable version rather than less reliable multi-versions.

  The two assumptions about the nature of fault fail-stop and transient are dated back to the early 1980's and they can be probably true for some relatively simple applications. But, these assumptions will simply not hold for modern distributed communication applications. Everyday experience with communication applications has shown that many (if not most) of the software faults are permanent and they are reproducible, but they require rare sequence of events to be activated. This can be explained with the fact that it is almost impossible and not realistic to test every path and combination in these large and complex applications. In some work on open-source applications (Apache web server and MySQL database)[Chandra00b], it has been found that deterministic faults are about 72-87% of the total number of faults. Another study on database management system [Chandra98] has found that 7% of the faults violate the fail-stop property. It should be mentioned, though, that mobile

communications applications are far more complex than the applications in these studies.

Note that we have only concentrated on the intrinsic limitations that nothing can be done about them. But, there are some other challenging problems that are difficult to resolve completely in real-world communication environment. For example, it is very difficult for active replication to preserve consistency across all replicas in the presence of non-deterministic behavior such as caused by operating system-specific calls, process/thread scheduling, timers, etc.

Chapter 3
Mobile Data Communication & Failure Recovery

In the previous chapter, we described the best known existing failure recovery techniques. In this chapter, we will investigate the communication characteristics of the mobile network infrastructure and analyze their implications on the existing failure recovery techniques. But, we look first at the overall architecture of mobile communication systems and then introduce TETRA packet data network as a concrete case study.

3.1 Overall architecture of mobile network

Mobile communications systems refer generally to any telecommunications system which enables a wireless communication when users are moving within the service area of the system. One of the main goals in the development of mobile communications networks is to provide new data communication services, such as packet data communication, and especially IP (Internet protocol) services. Because of its high efficiency, packet switched data services are expected to be the dominant type of communication in all modern mobile networks even to provide voice services (Voice over IP).

The overall architecture of packet switched mobile communications system is depicted in Figure 6 . The system can be roughly divided into 2 parts:

1. Mobile Stations (MS): mobile devices used to transmit and receive user data wirelessly. MS is composed of hardware and software, which implements a number of communication protocols. MSs can

move from one area (cell) to another and is still able to transfer data.

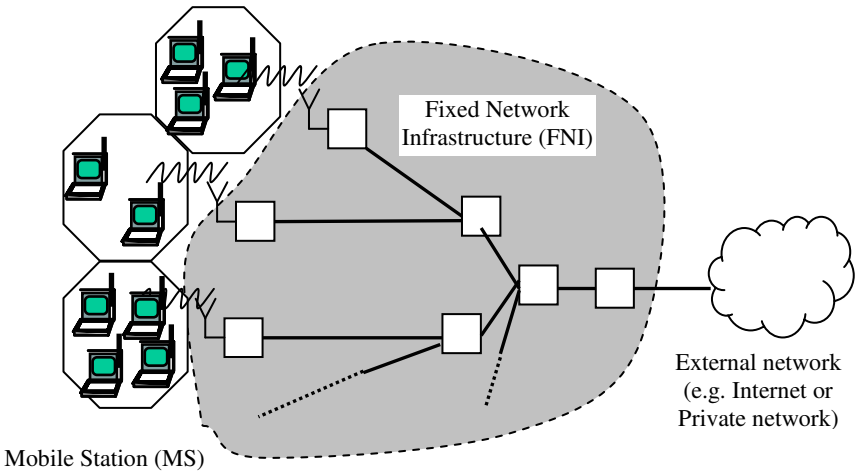

Figure 6: The overall architecture of the mobile communication system

2. Fixed Network Infrastructure (FNI): This is a fixed network consisting of base stations, routers, gateways, resource management, mobility management units, etc. that exist to support the operation of the wireless mobile stations. The FNI takes the overall coordination and control of the communication with the MSs and it uses peer-to-peer based protocols to achieve that.

The External network is an IP-based packet data network that contains the destination host requested by the mobile user. The External network can be the Internet or a customer private network e.g. LAN or X.25.

The fixed network connects the wireless mobile stations and the External network and consequently plays a critical role to the overall availability of the system. A failure in the FNI may disrupt the connection to hundreds or even thousands of MSs, this can be a very serious situation especially if the system is used in mission-critical applications such as in pubic safety, e.g. fire brigades, police forces and ambulance services.

A recovery from failure in mobile stations by using checkpointing and message logging has been investigated in the literature e.g. [Acharya94][Yao99][Pradhan96]. These literatures have been limited to simulation and never really studied the mobile communication protocols between MS and FNI. We also find it less important from the overall system availability point of view to focus on the recovery from failures in MS rather than FNI.

3.2 Mobile Communication Characteristics and their implications

Mobile data communication has several characteristics that must be taken into consideration when developing any failure recovery method. What these characteristics are and what implications do these characteristics have specifically for the failure recovery in mobile infrastructure is to be investigated in this section. These characteristics are as following:

- *Peer-to-peer client/server communication*: The communication pattern between an application in FNI and its peer application on MS is peer-to-peer; which means that the communication can be initiated by either side (MS or FNI). On the contrary, it is the client that always initiates communication in client/server model. However, the application in FNI provides service to many peer applications on MSs concurrently (one-to-many relationship) and it is also designed to be responsible for the overall

control, therefore the communication follows also client-server (master/slave) model at functional level. This mix of two communication models adds more complexity to the infrastructure applications. Furthermore, there is a coupling (dependency) between the application state in FNI and its peers on MSs resulting from use of the stateful communication in the design of mobile packet data services. The use of stateful protocols is the primary challenge for failure recovery process.

- *Real time communication*: Real time issue arises when there are actions that must be completed within a specified amount of time otherwise they become useless or even harmful after that. In this context, the entity that initiates requests should receive replies within a specified period of time otherwise timeouts occur. The real time aspects of mobile communication originate from both the end user application and the physical system. For example a user application (running in the application layer of OSI model) that monitors victims in the field and wirelessly sends information such as blood pressure and cardiac activity to the doctors in hospital needs to send this information and receive instructions instantly. In addition, real-time requirements are also imposed from the physical layer, for example by channel access schemes because data have to be transmitted in the assigned time slots.
- *High message rate*: The number of messages received and sent per unit time is high. Any single application in the FNI can easily send or receive many thousands of messages per minute. Therefore, any recovery technique that uses message logging has to deal with two particular problems i.e. overhead and storage.
- *Distributed service architecture*: The FNI is distributed over a large geographic area to provide mobility. It is normally that several applications running on different nodes cooperate together to complete a single service for an MS. This distributed architecture will affect the selection of fault

tolerance approach; it will for example favor distributed redundancy such as active/standby approach rather than cluster approach e.g. server pool.

- *Scarce radio resources*: The limited bandwidth of the air interface underlines the need of efficient communication between mobile stations and FNI. Therefore any extra communication to be brought by any recovery method should be carefully examined.

What these characteristics imply? The two characteristics real-time and peer-to-peer communication combined together implies that message logging and then replaying them- as done by log-based checkpointing and passive replication- by the applications in FNI after failure will not work because the peer applications on MSs may change their state (because of timeouts) and hence not able to deal with the outcome of these replayed messages.

What about taking coordinated checkpoints for applications in FNI and MSs? In this case when the application in the FNI rolls back after failure then all its peers on MSs need also to roll back. It is very difficult to imagine how complex the recovery protocol needed to manage this recovery and it will certainly exhaust the scarce radio resources.

The two characteristics real-time combined with high message rate will be a killer to any technique using voting mechanism such as N-version and active replication because the delay, which is caused by voting, is proportional with message rate, and a second problem is that voting of communication messages may require knowledge about their contents.

Finally, the distributed service architecture will strongly limit the use of active replication as it requires a significant increase in the number of hardware modules to run the various groups of replicas.

Table 3-1 summarizes the negative implications caused by the communication characteristics in the FNI.

	Coordinated checkpointing	Log-based checkpointing	Active replication	Passive replication	N-version programming
Peer-to-peer		÷		÷	
Real-time	÷	÷	÷	÷	÷
High message rate		÷	÷	÷	÷
Distributed service			÷		
Limited bandwidth	÷				

Table 3-1: Negative implications of FNI communication on recovery techniques

3.3 Requirements for failure recovery in mobile infrastructure

Let us start with some kinds of philosophy learned from experience. It is almost impossible to develop a complex application that is free from faults but it is possible through testing to reduce the number of faults to a level at which the application reliability is acceptable. No guarantees can be given to what errors caused by the remaining faults can do. Furthermore, client users do not care if the failure is caused by the server application, operating system or hardware error; they just require the service to be restored immediately.

Based on what we have studied and analyzed until now, we can point the following important requirements for failure recovery in mobile environment.

High availability: because of the distributed nature of the fixed network infrastructure, it is difficult to imagine a complete failure of the system. But in general, a level of five nines 0.99999 availability (i.e. 5 minutes downtime per

year) is considered to be quite good for mission critical communication. Since the system is composed of many nodes, the availability of a single node should be much better than five nines but this also depends on the importance of the node i.e. its failure impact on the end users. We estimate a full recovery time in the order of seconds to be good for a single node.

Low overhead without real-time drawbacks: The overhead – extra CPU utilization - that is caused by the presence of failure recovery method should be low so that application performance is not significantly affected. It is not sufficient with low overhead but it is also important that the overhead has no negative impact on real-time communication. By using checkpointing mechanism, for example, each process in the system is stopped every time a checkpoint is taken. Stopping processes causes time delays and consequently the application may fail to adhere to real-time constraints.

No assumptions on faults: The practical use of the recovery method will be significantly improved if no assumptions are made about the nature of faults, e.g. transient or fail-stop. It is for example a serious limitation in the case of active replication that a permanent (deterministic) software fault will take all replicated servers down and practically everything is lost.

Cost effective: Any proposed failure recovery method should be cost effective. Development of software systems for mobile communication infrastructure costs tens of millions of dollars. Solutions such as N-version programming that triple or even double the development cost have no chance to be adopted.

Chapter 4

Modeling Communication Applications

Although transparent failure recovery, which does not require any intervention on the part of the application or the programmer, is very tempting but unfortunately it can not give the solution to many real world systems. It is rather simplistic to treat all applications requiring fault tolerance as black boxes despite the differences in their functionalities, real-time constraints and performance requirements. As a result of using black box approach, it is necessary to put assumptions on the faults behavior for example to be transient, or even on the behavior of the application itself for example to act deterministically, which implies that application avoids using nondeterministic sources such as multithreading and timers. In contrast, we believe that understanding the communication applications behavior is a key factor in the success of the recovery techniques and this is the subject of this chapter.

4.1 OSI model

The OSI (Open Systems Inter-connection) layered model [ITU94] is the dominant model to develop mobile communication standards as well as to design and implement communication software systems. The OSI model provides a high level for system architecture and behavior. Figure 7 illustrates peer-to-peer communication in the layered OSI model. In a layered architecture, each layer comprises protocol entities that perform functions within the layer. The entities in the (N)-layer (and all layers below) provide *(N)-service* to the

(N+1) entities, through (N)-Service-Access-Points ((N)-SAP) at the boundary between the (N+1)-layer and (N)-layer. A protocol design for the (N)-layer defines both the (N)-service and the (N)-protocol. In the generic OSI model, peer (N)-protocol-entities virtually communicate by sending and receiving Protocol Data Units (PDUs), which consist of a header containing protocol

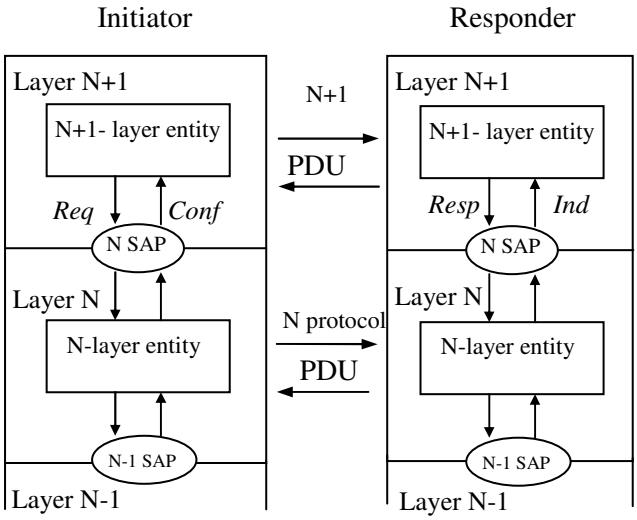

Figure 7: Protocol entities interaction in OSI model

control information and possibly user data. When the (N+1)-layer at the initiator needs to send a PDU to its peer, it sends it with a primitive *Request* to the lower (N)-layer. The (N)-layer at the responder sends a primitive *Indication* to deliver the PDU to the (N+1)- layer. The peer responds to the indication by sending a *Response* primitive to the lower layer. The lower (N)-layer at the initiator notifies the (N+1)-layer about the PDU delivery by sending a primitive *Confirm*. Therefore, the logical path for exchanging information is vertical, via SAPs. When a PDU passes a SAP, it becomes an SDU (Service Data Unit) at the receiving layer.

In peer-to-peer communication, the same protocol entity may become an initiator in one case and a responder in another. However, the functions that the protocol entity should have depend on whether it acts as service requester (initiator) or service provider (responder). The client/server model may also included under the OSI model, in the sense that an entity can be designed to provide services to a group of peer entities (one-to-many relationship) as it the case in mobile communication protocols.

In the context of OSI model and fault tolerance, Kenneth P. Birman [Birman96] has raised a very important question that, to the best of our knowledge, has been left without answer until now. *Can "well structured" distributed computing systems be built that can tolerate the failures of their own components?* In layering like the OSI one, this issue is not really addressed. The question is among the most important ones that will need to be resolved if we want to claim that we have arrived at a workable methodology for engineering reliable distributed computing systems.

In this dissertation, we claim that the OSI model is fault tolerance "friendly" and it provides a good overall framework to develop failure recovery method on.

The modular architecture of the OSI model fits well with the modularity key principle of fault tolerance, in the sense that protocol layer entities can be isolated from each other and the only interaction between them is through message passing. It is true that entities can still send corrupted messages to each other and thus open for error propagation but on the other hand it is quite possible for the entities to guard against this problem.

4.2 Modeling communication protocols by CEFSM

The OSI model, as mentioned in the previous section, provides an overall model for the distributed system behavior by defining the interaction mechanism between adjacent layers through SAPs and between peer layers by PDUs. In our

work to develop an effective failure recovery method for the FNI, we are further interested in a model that can describe in a sufficient degree the functional requirements of each protocol entity. The Communicating Extended Finite State Machine (CEFSM) is selected to formally describe the behavior of communication protocols entities. The CEFSM is used in a number of industrially significant specification techniques, such as SDL [ITU96] and UML [OMG02]. Our definition of CEFSM is different from that of [Byun02] in the sense that it is adapted to the OSI model to give it the ability to well model the complexity of the standard communication protocols.

Definition: A CEFSM is a 6-tuple (S, I, E, A, O, T) where

- S is a finite nonempty set of states, where one of these is initial state.
- I is a set of information elements with their types and initial values. The information elements are used by entities for coordination and control. Each information element i ($i \in I$) may have any number of bytes/bits and is shared by more than one entity (peer or adjacent). Examples of information elements are fields in the PDU header (e.g. sequence numbers), primitive parameters (e.g. request number) and constants defined by the protocol (e.g. maximum number of retransmissions and timer values).
- E is a finite nonempty set of input events. An input event e ($e \in E$) is one of the following three types:
    i. Receipt of an indication or confirm primitive from the next lower layer.
    ii. Receipt of a request or a response primitive from the next higher layer.
    iii. Receipt of an input signal that is triggered by, for example, timer expiration.

- A is a set of actions. This set may include various activities e.g. updating variables, incrementing/ decrementing counters, starting/stopping timers, queuing management, etc. An action a ($a \in A$) may update some information elements and/or it may also require input information elements for its execution, this is denoted as $a(i)$.
- O is a set of outputs. An output event o ($o \in O$) is one of following two types.
    i. Sending a request or a response primitive to the next lower layer.
    ii. Sending an indication or confirm primitive to the next higher layer
- T is a set of state transitions

$$(t: (s_{\text{current}} , e(\Psi)) \rightarrow ([a],[o], s_{\text{next}})$$

Where t is a mapping from each state-event pair (s_{current} ,e) to a corresponding action set, output set and next state s_{next}. The event e is associated with an optional predicate Ψ which is a condition that decides the selection of the next state. The predicate Ψ has the following form: $i \sim c$, where $\sim \in \{<, >, ==, \neq\}$ and $c \in \mathbb{N}$

The action and output sets contain zero or more elements, in other words the state-event pair may or may not trigger any action or send output. Note that CEFSM is deterministic because the selection criterion to make the transition for each state-event is clearly defined.

It is in place to give a more clear definition of some terms. We use the term *entity* to mean a layer or a software process/thread that is actually an implementation of the functions that are defined for a given layer. The functionality provided by a layer is formally expressed by its set of

transition *T*. The term *state information* includes both the set of states *S* and the set of information *I* of the entity. Finally, an *application* includes one or more entities.

4.3 Case Study: TETRA Packet Data

TETRA packet data protocol [ETSI03] is developed by European Telecommunication Standards Institute (ETSI) to provide wireless data service that satisfy the most demanding mobile radio users, particularly users working in public safety, e.g. police, fire brigade and ambulance service. TETRA packet data protocol has many similarities with the General Packet Radio Service (GPRS) that has been built on GSM to provide IP packet data services. We use TETRA packet data as a concrete case study to firstly apply the CEFSM model and secondly to use it later to build an experimental testbed for evaluation of our proposed recovery method. The selection of TETRA packet data does not restrict the applicability of the proposed method to TETRA; it may be well used on other protocols for example GPRS or UMTS (Universal Mobile Telecommunications System).

TETRA packet data is built on top of the basic TETRA radio link protocol stack and provides service mechanisms to convey different higher layer protocols. The network layer protocols supported by the TETRA packet data include Internet Protocol (IP) versions 4 and 6. Thus the TETRA packet data extends the TETRA network to act as an IP subnet in the mobile IP scheme, which enables application programmers to build their applications in a well standardized environment. Figure 8 illustrates the protocol stacks of the TETRA packet data when an application using the IP protocol is located in a mobile station MS. The fixed network infrastructure (also referred to as Switching and Management Infrastructure in TETRA terminology) communicates over an air interface R_0 with a TETRA mobile station.

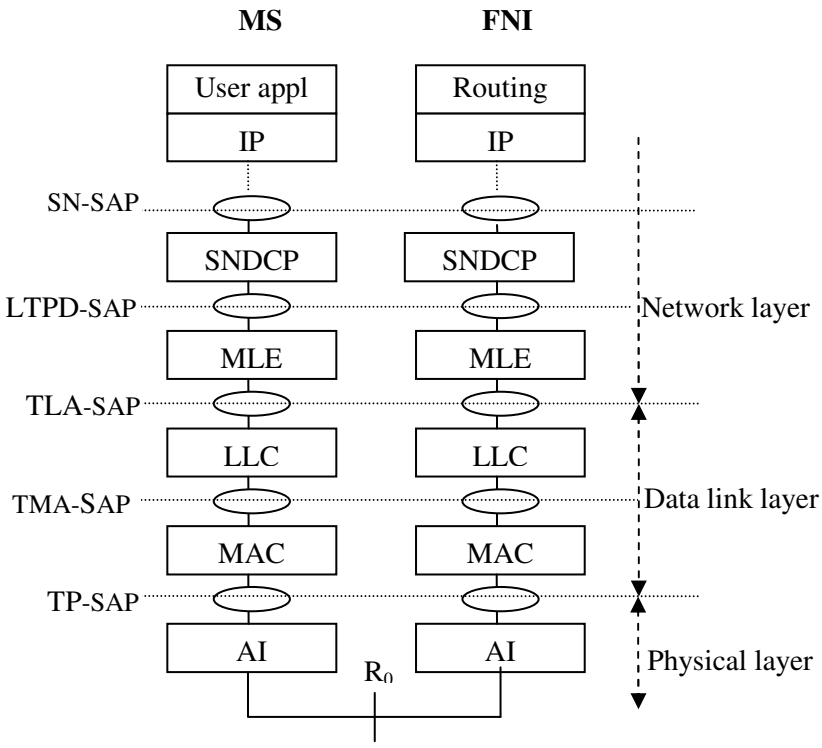

Figure 8: TETRA Packet Data Protocol stack

The TETRA packet data protocol stack provides the specifications for a number of protocols that cover the physical layer, the data link layer, and the network layer of the OSI model.

We briefly describe the TETRA Packet Data Protocol stack, starting from the highest layer of the stack and working our way downward.

- SubNetwork Dependent Convergence Protocol (SNDCP): This stateful protocol is used to negotiate and maintain PDP (Packet Data Protocol) context between MS and FNI. Before any user IP packets can be conveyed by the SNDCP layer, it is necessary for the MS to successfully negotiate a PDP context with the infrastructure in order to gain access to SNDCP services. PDP context activation involves the negotiation of a PDP address (e.g. an IPv4 address) and other parameters (e.g. timers' values) to be used

during data transfer. Furthermore, control of PDP data transfer, packet data channel handling and data compression is also performed in this layer. The SNDCP provides services to its user at SN (Symbol Number) SAP.

- Mobile Link Entity (MLE): This layer is used to manage mobility and radio resources for the higher SNDCP layer. The MLE layer performs surveillance of the quality of the radio communication path based on information received from the MAC layer. MLE shall also report any loss or break of the path, for example due to cell change. The MLE entity provides services to SNDCP via LTPD (Link Entity TETRA product Data) SAP.
- Logical Link Control (LLC): The LLC layer provides two types of logical links, basic link for connectionless services and advanced link for connection-oriented services. Basic link is used for short messages like signaling messages, while advanced link is used for long messages data transfer that requires some type of QoS. This layer offers segmentation of long messages, retransmission, and error control using frame check sequence. LLC entity provides services to MLE at TLA (Type Identifier on Accept) SAP.
- Medium Access Control (MAC): This layer is responsible for channel access, MAC uses TDMA (Time Division Multiple Access) access scheme with four physical channels (timeslots) per carrier. MS-MAC layer uses random access based on slotted ALOHA procedures to initiate transaction and reserved access for further processing in order to achieve higher channel throughput. MAC layer performs other functions such as channel coding, forward error correction, measurement of the signal quality and encryption over the air interface. The MAC services are accessed at the TMA (Transmit Multiple Access) SAP.

- Air Interface (AI): This is the physical layer, which is responsible for modulation/ demodulation, frame synchronization and power control. The services of the AI are accessed at TP (Traffic Physical channel) SAP.

4.3.1 CEFSM model for SNDCP protocol

We select the FNI SNDCP protocol entity in FNI to demonstrate how a CEFSM model can be developed. The SNDCP protocol is used by different mobile communication standards e.g. GPRS. The SNDCP protocol services, however, should exist in every mobile communications system that provide wireless data services.

We give first a description of the SNDCP protocol in accordance with the standard. SNDCP protocol maps a network-level protocol, such as IP, to the underlying wireless protocols. SNDCP also controls packet data transfer between the MS and FNI. An MS can be in Idle, Standby or Ready state depending on its current activity. In the Idle state, MS is not reachable; no data transfer to and from the FNI is possible. In order to transfer data, the MS shall perform a PDP context activation procedure with its peer in the infrastructure. After completing a successful PDP context activation, the MS enters Standby state. A Standby timer associated with Standby state to control the time an MS retains SNDCP services after data service inactivity. The purpose of the Standby timer is to work as a fallback timer to delete PDP contexts when they remain unintentionally undeleted and thus having better resource utilization. The Standby timer is in the range of hours and is started on entry to Standby state. In the Ready state the MS may receive and transmit data. MS enters Ready state when it is granted a data channel. A Ready Timer associated with Ready state to control the time an MS may remain inactive on data channel after data service activity. The Ready timer is in the range of seconds and is started on entry to Ready state. Table 4-1 shows our CEFSM model for the SNDCP

$S = \{s_1, s_2, s_3\}$	{ Idle, Standby, Ready}
$I = \{i_1, i_2,$ $i_3, i_4,$ $i_5, i_6\}$	{create_context_response_status, ip_addr, data_transmit_response_status , ms_location, standby_timer_value, ready_timer_value}
$E = \{e_1,$ $e_2,$ $e_3,$ $e_4,$ $e_5, e_6,$ $e_7, e_8,$ $e_9, e_{10}\}$	{ACTIVATE_PDP_CONTEXT_DEMAND PDU in ind., Create_PDP_Context response, DEACTIVATE_PDP_CONTEXT_DEMAND PDU in ind, DATA_TRANSMIT_REQUEST PDU in Ind., DATA PDU in Ind , Data_Packet request , Transmission_Report indication, RECONNECT PDU in Ind , Standby_timer_expire, Ready_timer_expire}
$A = \{a_1, a_2, a_3,$ $a_4, a_5, a_6, e_7\}$	{ Set_value, Cretate_ms_record, , Delete_ms_record, Stop_timer, Start_timer, enqueue_packet, dequeue_packet }
$O = \{o_1, o_2,$ $o_3,$ $o_4,$ $o_5,$ $o_6,$ $o_7,$ $o_8, o_9,$ $o_{10},$ $o_{11},$ $o_{12}, o_{13}\}$	{Create_PDP_Context request, Delete_PDP_Context request, ACTIVATE_PDP_CONTEXT_ACCEPT PDU in Resp., ACTIVATE_PDP_CONTEXT_REJECT PDU in Resp., DEACTIVATE_PDP_CONTEXT_ACCEPT PDU in Resp, DEACTIVATE_PDP_CONTEXT_DEMAND PDU in Req, DATA_TRANSMIT_RESPONSE PDU in Resp, Data_Packet indication ,DATA PDU in Req, DATA_TRANSMIT_REQUEST PDU in Req, Packet_Delivery_Status indication, PAGE_REQUEST PDU in Req, END_OF_DATA PDU in Req}
$T = \{t_1,$ $t_2,$ $t_3,$ $t_4,$ $t_5,$ $t_6,$ $t_7,$ $t_8,$	{ $(s_1, e_1) \rightarrow ([o_1], s_1)$ $(s_1, e_2(i_1 == \text{“Accepted”})) \rightarrow ([a_2, a_1(i_2), a_5(i_5)], [o_3], s_2)$ $(s_1, e_2(i_1 == \text{“Rejected”})) \rightarrow ([o_4], s_1)$ $(s_2, e_3) \rightarrow ([a_4(i_5)], [o_2, o_5]], s_1)$ $(s_2, e_{10}) \rightarrow ([a_3], [o_2, o_6] s_1)$ $(s_2, e_4) \rightarrow ([a_2(i_3), a_4(i_5), a_5(i_6)], [o_7], s_3)$ $(s_2, e_6(i_4 \neq \text{“Known”})) \rightarrow ([a_6], [o_{12}], s_2)$ $(s_2, e_6(i_4 == \text{“Known”})) \rightarrow ([a_6], [o_{10}], s_2)$

t_9,	(s_2 , e_7)→([a_7,$a_5(i_6)$], [o_9], s_3)
t_{10},	(s_3 , e_7)→([$a_4(i_6)$,$a_5(i_6)$], [o_{11}], s_3)
t_{11},	(s_3 , e_6)→([o_9], s_3)
t_{12},	(s_3 , e_8) →([$a_1(i_4)$, $a_4(i_6)$, $a_5(i_5)$], s_2)
t_{13} ,	(s_3 , e_5) →([$a_4(i_6)$,$a_5(i_6)$], [o_8], s_3)
t_{14} }	(s_3 , e_{10}) →([$a_5(i_5)$], [o_{13}], s_2) }

Table 4-1: CEFSM model for the SNDCP entity in FNI

protocol entity in FNI. The formats of the PDUs exchanged by SNDCP peer entities are listed in Appendix A. CEFSM is usually represented graphically by a state transition diagram (STD), a directed graph whose vertices correspond to states and whose edges correspond to transitions. Figure 9 shows the STD of the FNI SNDCP protocol entity. Each state is represented by a circle, and the initial state has a double circle. Transition that does not lead to a new state is represented by an arc that points to itself.

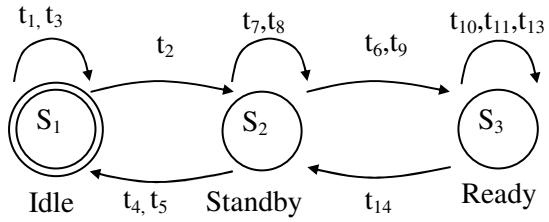

Figure 9: STD of SNDCP entity in FNI

A description of the SNDCP state transitions is given below:

t_1: On reception of an ACTIVATE_PDP_CONTEXT_DEMAND PDU [Table A-1] in an indication primitive (event e_1) from the lower layer MLE at state

Idle, the SNDCP entity in FNI sends Create_PDP_Context request to the upper layer (output o_1).

t_2 : Upon receiving Create_PDP_Context response from the upper layer (e_2) with response status set to "Accepted" (i_1 == "Accepted") to indicate that PDP context is created successfully, create MS record (action a_2), set the MS IP address to the received value $a_1(i_2)$, send ACTIVATE_PDP _CONTEXT_ACCEPT PDU[Table A-2] in a response to MLE (o_3) , start Standby timer $a_5(i_5)$ and finally enter the Standby state

t_3 : Upon receiving Create_PDP_Context response from the upper layer (e_2) with response status "Rejected" (i_1 == "Rejected") to indicate that PDP context is not created then send ACTIVATE_PDP_CONTEXT_REJECT PDU[Table A-3] in a response to MLE (o_4).

t_4: On reception of a DEACTIVATE_PDP_CONTEXT_DEMAND PDU [Table A-7] in an indication primitive (e_3) from MLE layer at Standby, send DEACTIVATE_PDP_CONTEXT _ACCEPT PDU [Table A-8] in a response to MLE (o_5) , send Delete_PDP_Context request to the upper layer (o_2), delete MS record a_3, stop Standby timer $a_4(i_5)$ and finally enter the Idle state.

t_5 : Upon expiry of Standby timer (e_9), send DEACTIVATE_PDP_CONTEXT _DEMAND PDU in a request to MLE (o_6) , send Delete_PDP_Context request to the upper layer(o_2), delete MS record a_3 and enter the Idle state.

t_6 : On reception of a DATA_TRANSMIT_REQUEST PDU[Table A-5] in an indication from MLE (e_4), send DATA_TRANSMIT_RESPONSE PDU[Table A-6] (with Accept) in a response to MLE (o_7), stop Standby timer $a_4(i_5)$, start Ready timer $a_5(i_6)$ and finally enter the Ready state.

t_7 : Upon receiving Data_Packet request containing IP data packet from the upper layer at Standby state (e_6) and MS location is not known ($i_4 \neq$ "Known") , queue the IP packet a_6, send PAGE_REQUEST PDU[Table A-9] in a request to MLE (o_{12}).

t_8 : Upon receiving Data_Packet request from the upper layer at Standby state (e_6) and MS location is known (i_4 == "Known") , queue the IP packet a_6, send DATA_TRANSMIT_REQUEST PDU in a request to MLE (o_{10}).

t_9 : Upon receiving Transmission_Report indication from the MLE (e_7) at Standby state, remove the IP packet/s from the queue a_7, encapsulate Packet/s in DATA PDU/s[Table A-4] and send as request primitive to MLE (o_9), start Ready timer and enter Ready state.

t_{10} : Upon receiving Transmission_Report indication from the MLE (e_7) at Ready state, send Packet_Delivery_Status indication containing transfer status to the higher layer (o_{11}), and restart (stop and then start) Ready timer.

t_{11} : Upon receiving Data_Packet request containing IP data packet (e_6) from the higher layer at Ready state, place the IP packet in a DATA PDU and send the PDU in a request primitive to MLE (o_9).

t_{12} : Upon reception of a RECONNECT PDU[Table A-10] in an indication from MLE (e_8) - indicating that MS changed cell-, update the new location $a_1(i_4)$, stop Ready timer $a_4(i_6)$, start Standby timer $a_5(i_5)$ and enter the Standby state.

t_{13} : On reception of a DATA PDU -containing IP packet data- in an indication from the lower layer MLE (e_5), restart Ready timer and send Data_Packet indication to the upper layer (o_8).

t_{14} : Upon Ready timer expiration (e_{10}), send End_Of_Data PDU[Table A-11] in a request to MLE (o_{13}), start Standby timer $a_5(i_5)$ and then enter the Standby state.

As it can be noticed, by using our OSI-adapted CEFSM, communication protocols can be modeled with a high level of detail to include any of the specifications in the standard. That also means that the CEFSM model describes the behavior of the protocol entity very realistically. Finally, a CEFSM model for the SNDCP entity at MS can also be developed in the same way according to the standard specifications. The MS SNDCP entity, however, has some extra states e.g. to handle cell change situation.

Chapter 5

State Transition Based Recovery (STBR)

Armed with the CEFSM model, we are ready to attack our main goal to develop a recovery technique that can improve the service availability in the mobile infrastructure. The proposed technique should be realistic enough to deal with real world programming faults, relatively easy to understand and implement, and cost effective. We call our recovery method *state transition based recovery* (STBR) because it is based on CEFSM model which in turn is based on the traditional state transition model.

5.1 Objective & assumptions

STBR failure recovery method should be able to tolerate software and hardware faults without any assumption on the nature of faults. The method should work in the mobile environment and has no negative impact on the real-time communication. The faulty entity in the FNI has to resume communication after a failure in a way that hides the failure from all its peer entities on MSs. In other words, the peers should always receive input events in accordance with their protocol specifications.

Although there are no assumptions on the nature of faults e.g. to be transient or fail-stop, there are two prerequisites that need to be satisfied in order for the STBR method to deliver the promised high availability:

I. Error detection mechanism: We assume that there are mechanisms to quickly detect the failure and either to restart the faulty entity on the

active node or to immediately run it on a redundant node. Fault detection is an important part of building fault tolerant systems, but it is beyond the scope of this work. There are two main approaches to perform fault detection. Firstly, by monitoring *locally* the application for example by letting the entity kicks a watchdog timer as long as it is running to indicate that it is still in operating state [Mahmood88]. Secondly, by *remotely* sending periodic heartbeats and expecting responses from that entity [Han99]. These techniques can detect failures caused by hardware faults or coding faults which cause the application to crash or hang. But design faults - where the application works correctly from software point of view but fails to provide correct service according to the specifications – are rather difficult to detect.

II. Software reliability: The software applications to be recovered need to be reasonably reliable before becoming high available, an application that fails once every day on average is not reliable enough. In other words, it is desirable that the activation rate of the remaining software faults which can lead to failures is relatively low. What that means in practice? It means that these remaining faults are activated, for example, by rare scenarios that are not tested or slow memory leak. Software reliability can be achieved by a good test plan that includes different types of tests e.g. unit test, integration test, system test, etc. Testing complex software systems such as mobile communication is extremely difficult and time consuming due to the large number of scenario and test cases to be considered. The hardware reliability is not considered because it is very high nowadays.

5.2 STBR Approach

A well designed protocol should always consider the disturbances that may occur in the communication environment between clients and servers or more generally between sender and receiver such as time delays , data loss, duplication and out of order. These disturbances in the communication path are coped with by the protocols, mainly as exceptional cases that shall be handled by both sender and receiver. Our intention is to utilize that for the recovery of the entities in FNI in such a way that a failure is experienced as a disturbance by the client entities on MSs. Thus the basic idea is to handle the failure as a disturbance that the application should recover from by itself.

In the STBR approach, we seek to conform both to the key principles of building fault tolerant system and to the specific requirements of mobile environment. The STBR method applies the following set of principles to reach the objective:

- *Restarting faulty entity*: Restarting the faulty entity as a first step of recovery process is a secure way to ensure that the entity is free from the error that caused its failure. All existing techniques that try to get the latest saved state, and then reach the same internal state as if fault has not occurred, have no guarantee that the error is cleared from the saved states. Consequently, there is a probability that the entity fails again short after its recovery. In case of permanent software fault there is still a risk that the entity using STBR do fail again after restart, but the probability is very low because the rare situation which activated the fault that lead to the failure has to re-occur. Why this should not be also true for rollback techniques and passive replication? These techniques intentionally seek to repeat the exact “faulty” pre-failure execution during their recovery while STBR starts the execution from the beginning and focus on service recovery. Restarting from the beginning is a secure remedy

against transient faults and the best solution against permanent faults, however, it needs to be followed by a fast recovery.

- *Model-based recovery*: The STBR recovery is based on the behavior of the application represented by the CEFSM model. We will use our case study from previous chapter to explain this. Assume that the SNDCP entity in FNI crashed while servicing its peer clients on MSs and then get restarted. Normally, the FNI SNDCP entity will start operating from the initial state (S_1), so if the SNDCP entity should function correctly then all peer clients

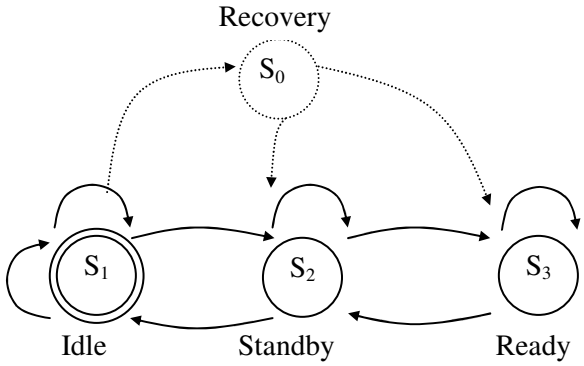

Figure 10: STD of FNI SNDCP entity extended with Recovery State

that are not at initial should go back to initial state. Thus, without any recovery method clients not at initial state need to return to initial state and then redo some work to reach their pre-failure state. How can STBR fix this situation?

According to the CEFSM model, if the protocol entity processes the set of events *E* in compliance with the specified set of transitions *T*, then it behaves correctly. Figure 9 shows actually how this is achieved for an FNI SNDCP entity that does not fail. Figure 10, on the other hand, is extended with Recovery state (S_0) to include failure situation. After a failure, the FNI SNDCP entity restarts as usual and assumes by default that all its peer entities at initial state (S_1). The FNI SNDCP entity becomes inconsistent once it receives an input event related to a peer that is not at initial state. To solve inconsistency, the

entity enters Recovery state(shown as dashed circle), finds out what is the current correct state of that peer, returns to the consistent state (any state other than initial) and finally executes the corresponding state transition. The Recovery state is entered from Initial state only once and only for the MS entities that need recovery. The FNI entity needs also to be aware about its failure in the previous execution before moving to Recovery state. We explain later in this chapter in details the recovery mechanism.

To achieve the above mentioned recovery steps, the FNI SNDCP entity should know at any time the current state $s_{current}$ and information elements I of every peer entity. Therefore, the FNI SNDCP entity needs to save state information of every peer entity during failure-free execution and to use them after failure in order to process all input events correctly. In the next section a recovery protocol is developed to save/retrieve state information to/from a stable storage.

- *Autonomous recovery*: The faulty entity should be able to autonomously do self-recovery without involvement from either peer or adjacent entities. This principle will ensure that no modification is needed for MSs. Furthermore, the restart of a faulty entity does not require the restart of any other entity. Finally, it will not be necessary that all entities in system need to be built with the STBR.
- *Active/standby redundant system*: Active/standby redundancy is an effective technique to prevent single point of failure in a distributed system. In case of a hardware fault, the faulty entity should be able to immediately start on the redundant standby node and resume the service to its peer entities on MSs. This ability can also be used for software faults in the operating system to avoid delay caused by reboot process. Active/standby approach is cost effective because the standby node can be used as a backup for more than a single node (N+1 redundancy) assuming that all active nodes running the same operating system.

5.3 Recovery protocol

To enable the FNI entities to easily save/retrieve their state information, we have developed a protocol that uses the client server architecture, see Figure 11. Every entity in FNI that uses STBR method sends states *S* and information elements *I* during failure-free execution to a server task referred to as State Information Saver (SIS) for storing, and then continues its execution (non-blocked mode). The SIS runs on a separate hardware waiting for requests in a blocked mode, requests that need responses will always be processed before any others in order to minimize response time. The SIS adds timestamp and saves the information on a stable storage e.g. non-volatile RAM or hard disk. The entity should first register itself to the SIS before it can start saving state information. The client side of the protocol (FNI) is implemented as a user level library of C functions that can be linked with the FNI entities. Some of the user functions provided by the library are listed below:

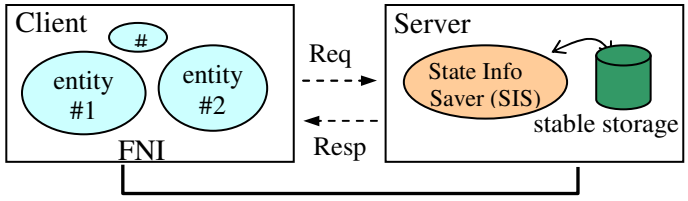

Figure 11: Recovery protocol using client/server model

- Entity_Register (entity_id): This function sends a request message of type REGISTER (refer to) to the SIS to register the entity identified by *entity_id*. Every entity must call this function once it starts up. The entity will receive a response message from SIS with one of the following response code "Ok", "Not_ok" or "Aleardy-registered", the latter response code indicates that the entity has not exited normally from last run (possible crash).

- Entity_Deregister (entity_id): This call sends a request message of type DEREGISTER to the SIS to deregister the entity identified by *entity_id* field. This call must always be executed before the entity exits. Failing to do deregistration will result in receiving "Already_registered" response status in

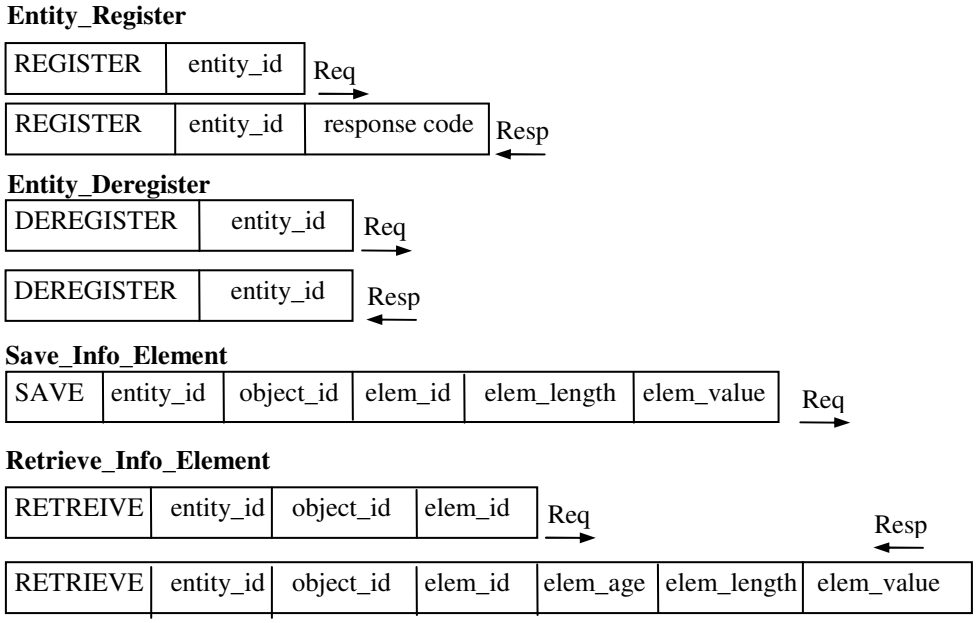

Figure 12: Format of requests and responses used in recovery protocol

the next registration. The entity will receive a response message of type DEREGISTER from SIS task to acknowledge deregistration.

- Save_Info_Element (entity_id, object_id, elem_id, elem_length, elem_value): This call sends a request message of type SAVE to save or update state information element that is identified by *elem_id* and of length *elem_length* in bytes. The *elem_value* contains the raw data information. The *object_id* specifies the object that the information element belongs to, for example *object_id* could specify the MS identification number.
- Retrieve_Info_Element (entity_id, object_id, elem_id): This call sends a request message of type RETRIEVE to retrieve information element for entity

with id number *entity_id.* The information element has id number *elem_id* and belongs to MS identified by *object_id*. A response message is received from SIS contains *object_id*, *elem_id*, *elem_age*, *elem_length* and *elem_value* fields. *elem_value* contains the requested information element and *elem_age* is the time elapsed since last save. The information element age is useful for the information that is time dependent.

The advantages of using client-server model in the recovery method are following:

I. The failure-free overhead caused by STBR recovery method will be low because entities need only to send requests while the SIS task executes save function and search algorithm.
II. The entity will not be suspended during failure-free execution as it is the case in checkpointing mechanism and thus no negative impact on time-critical communication.
III. Good scalability, as the system gets larger i.e. number of entities in FNI is increased, it should be easy to add more SIS tasks to serve them.

5.4 Mechanism

The mechanism of the STBR method will be illustrated for FNI SNDCP entity by using MSC (Message Sequence Charts) [ITU-T recommendation Z.120]. MSC is one of the most popular languages used in telecommunication to show interaction between protocol entities. MSC diagrams describe the behavior of the system in the form of message flows. We will explain how the recovery scheme acts during failure-free execution e.g. during PDP context activation and data transfer scenarios, and how recovery is executed after failure. The reader should refer to the SNDCP developed CEFM model [section 4.3.1] in order to completely understand these scenarios.

5.4.1 STBR during failure-free execution

Every FNI entity that uses STBR method must call Entity_Register once it starts to register itself to SIS. After receiving an "OK" response, the entity saves it state information during its execution as illustrated in the following scenarios for SNDCP entity.

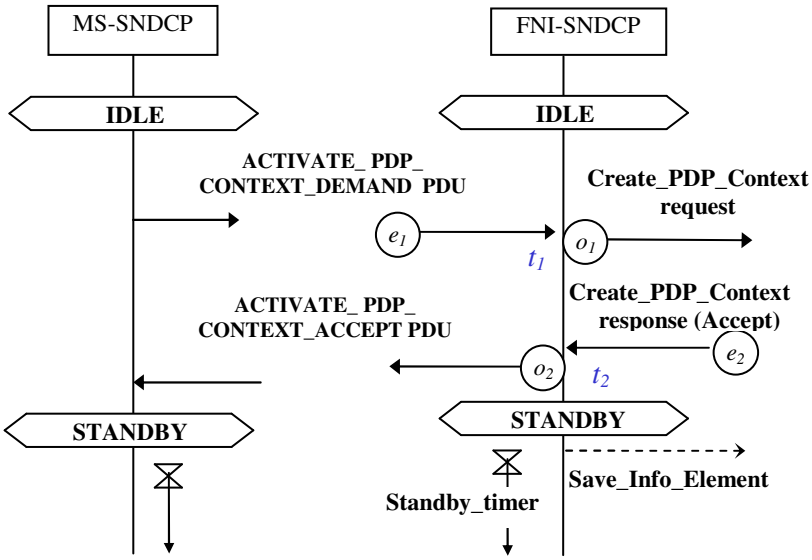

Figure 13: MSC showing STBR during successful PDP context activation

The MSC diagram in Figure 13 shows how STBR acts during a successful PDP context activation. Dashed arrows are used to show the communication with the SIS, while solid arrows show the communication with adjacent layers, and the flow of time occurs downward. The SNDCP entity in FNI and all peer entities on MSs will begin at initial state IDLE. When MS needs to start wireless data services, the MS SNDCP sends an ACTIVATE_PDP_CONTEXT_DEMAND PDU to FNI SNDCP. The FNI SNDCP receives ACTIVATE_PDP_CONTEXT_DEMAND PDU in an indication primitive from the lower layer MLE (input event e_1), so it executes

transition t_1 where Create_PDP_Context request is sent to the upper layer (output o_1). Upon receiving Create_PDP_Context reponse (e_2) indicating that PDP context is created, transition t_2 is executed where ACTIVATE_PDP_CONTEXT _ACCEPT PDU is sent in a response primitive to MLE (o_2), Standby state is entered , Standby timer is started and finally Save_Info_Element_Req is sent to save the new state and other information elements such as received IP address and timer values (i_2, i_5, i_6) . When MS SNDCP receives ACTIVATE_PDP_CONTEXT_ACCEPT PDU, it enters STANDBY state and starts Standby timer.

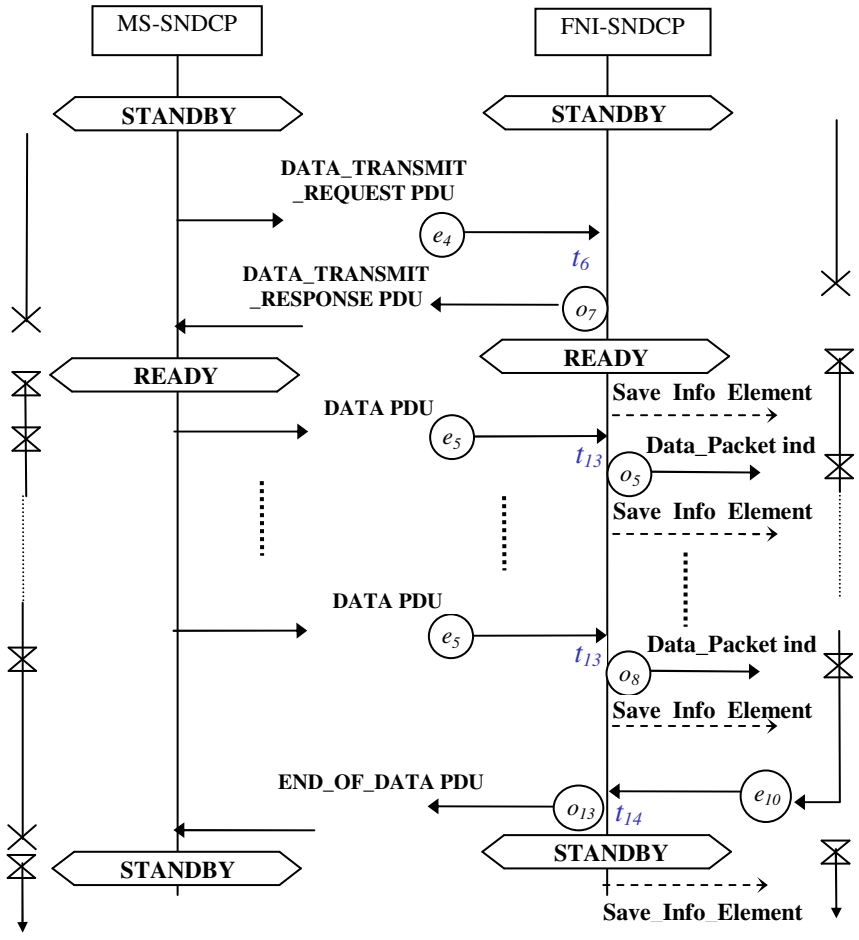

Figure 14 MSC showing STBR during uplink data transfer

Figure 14 shows how the STBR acts during uplink data transfer (from MS to FNI). In this scenario the MS is at Standby state and has user data to send, so MS SNDCP entity sends DATA_TRANSMIT_REQ PDU to FNI. When the FNI SNDCP receives DATA_TRANSMIT_REQ PDU in an indication (e_4), it executes transition t_6 where DATA_TRANSMIT_RESPONSE PDU (with Accept) is sent in a response to MLE (o_7), Standby timer is stopped $a_4(i_5)$, Ready timer is started $a_5(i_6)$, Ready state is entered and finally Save_Info_Element is called to save the new state. The MS SNDCP receives DATA_TRANSMIT_RESPONSE PDU with a granted data channel and begins transmitting the first IP packet in a DATA PDU. The FNI SNDCP receives DATA PDU and executes transition t_{13} where it forwards the IP packet in Data_Packet indication to the higher layer (o_5), restart the Ready timer, and finally call Save_Info_Element to update the Ready state. Although transition t_{13} re-enters the same state (Ready), however, calling Save_Info_Element will make it possible after failure to resynchronize with the peer by calculating the time passed since state was last re-entered (*elem_age* is used). This is useful when the state is associated with a timer. MS SNDCP continues to send DATA PDUs to the FNI SNDCP entity until no more data remains. When FNI SNDCP Ready timer expires (e_{10}), transition t_{14} is executed where END_OF_DATA PDU is sent in a request to MLE (o_{13}), Standby state is entered, Standby timer is started and finally Save_Info_Element is called to save the new state.

Figure 15 shows how the STBR acts during downlink data transfer (from FNI to MS). In this scenario FNI has data (e.g. originating from another MS or dispatcher center) to send to an MS that is at Standby and whose location is known. On the reception of the first packet in Data_Packet request from the upper layer (e_6) the FNI SNDCP entity executes state transition t_7 where it queues the packet and then sends DATA_TRANSMIT_REQUSET PDU in request primitive to MLE entity. When the PDU is completely transmitted on the air, the FNI SNDCP receives Transmission_Report indication (e_7) and

executes state transition t_9 where the queued packet is placed in DATA PDU and sent in a request to MLE (o_9), Ready state is entered, Ready timer is started and finally Save_Info_Element is called to save the new state. After

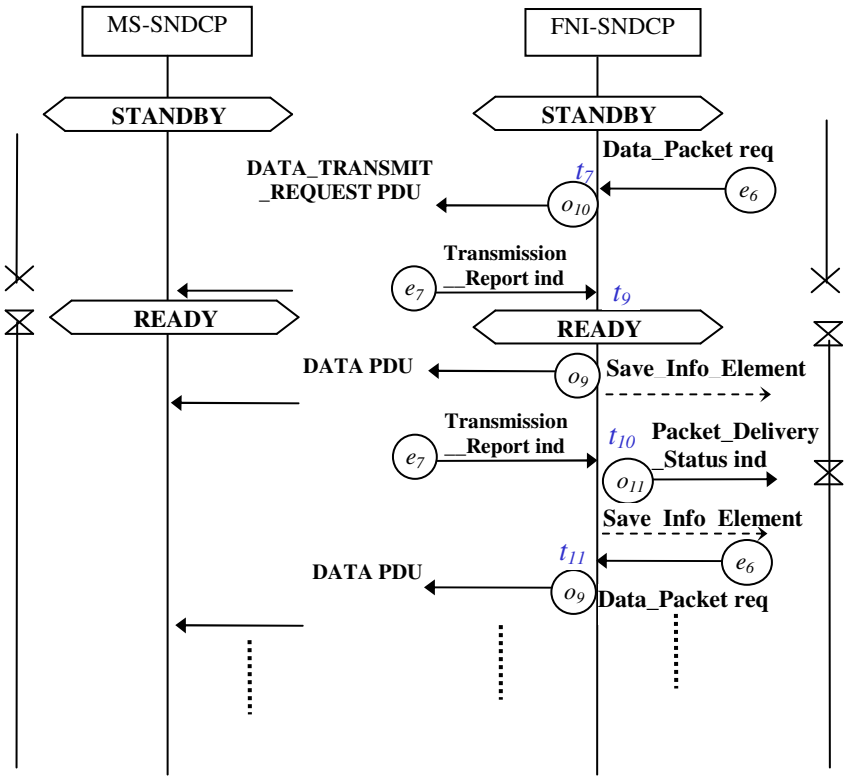

Figure 15 MSC showing STBR during downlink data transfer

transmitting the DATA PDU on the air the FNI SNDCP receives Transmission_Report indication and executes state transition t_{10} where Packet_Delivery_Status indication is sent to inform the upper layer about the transfer of the packet, Ready timer is restarted and finally Save_Info_Element is called to update the state. The upper layer continues to send the remaining packets in Data_Packet requests to the FNI SNDCP entity in the same manner.

Let us for a moment compare state information saving by STBR with state saving by checkpointing mechanism. In recovery techniques that use checkpointing there are two factors to struggle with during failure-free execution, and they need to be considered carefully. The two factors are the comprehensiveness of the process state being saved and the frequency at which the process state to be saved. In other words, how much process information should the checkpoints include to fully describe the state of the application and how often checkpoints to be taken. In deciding the degree of comprehensiveness and frequency to be used, there is a tradeoff between the amount of lost work and the performance overhead. This is not an easy task to solve without knowledge about the application to be recovered, therefore most checkpoint-based recovery schemes let the application programmer determines when to take checkpoints. However, in STBR as we can see from the scenarios these are nicely determined by the STBR mechanism. The STBR saves only the necessary state information and at the right time.

5.4.2 STBR during failure recovery

Once the faulty FNI entity restarts after failure, it calls Entity_Register as usual, but it receives "Already_registered" response this time from SIS because it did not call Entity_Deregister as a result of failure. That response can be used to make the entity aware of its previous failure. The entity can also get automatically informed about its previous failure by error detection utilities. In this section different SNDCP scenarios are used to show how the STBR performs recovery after failure.

Figure 16 shows an MSC diagram of recovery procedure for an MS initiated communication. The MS was at Standby state when the FNI SNDCP entity crashed. In this scenario the MS needs to send data, so MS SNDCP entity sends DATA_TRANSMIT_REQ PDU to FNI. The FNI SNDCP entity receives DATA_TRANSMIT_REQ PDU in an indication (e_4) at initial state Idle, but

since the state-event pair (e_4, s_1) is inconsistent with T (i.e. does not belong to the set of state transitions T) and in addition to its knowledge of the previous failure, FNI SNDCP enters Recovery state and calls Retrieve_Info_Element to

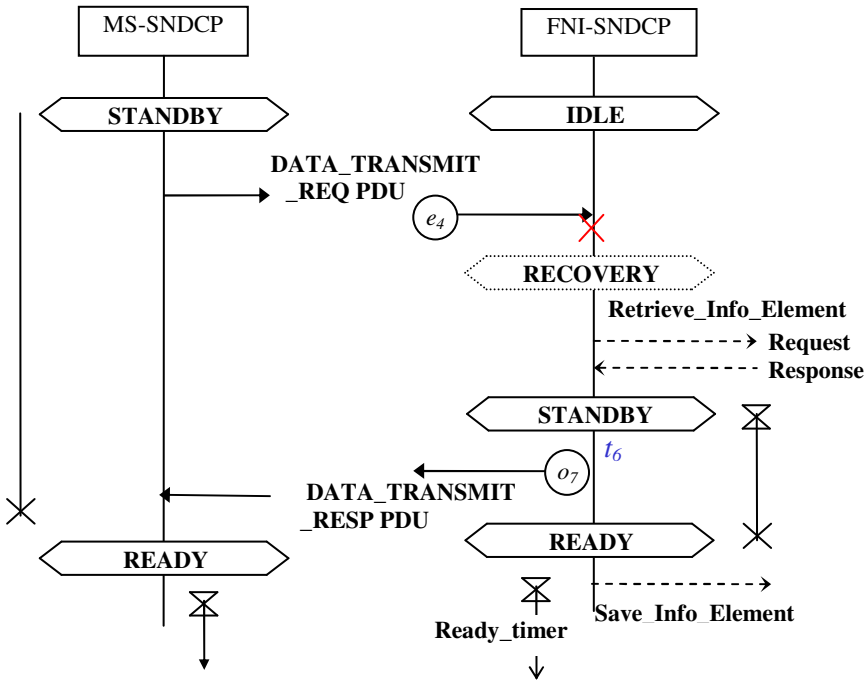

Figure 16: Recovery of an MS initiated communication (Standby)

retrieve the saved state and information elements (i_3, i_5, i_6) from SIS for the MS in concern. Based on the retrieved information, the FNI SNDCP enters Standby state and starts Standby timer with the calculated value (i_5 - *elem_age* of Standby state) to achieve synchronization with the running Standby timer in MS SNDCP. The FNI SNDCP will execute the state transition t_6, where DATA_TRANSMIT_RESPONSE PDU is sent in a response (o_7) to the next lower layer, Standby timer is stopped, Ready timer is started and finally the Ready state is entered. The MS SNDCP receives

DATA_TRANSMIT_RESPONSE PDU, starts Ready timer and enters Ready state. The consistency is now fully restored between the FNI SNDCP and MS SNDCP and the communication between them can continue as normal.

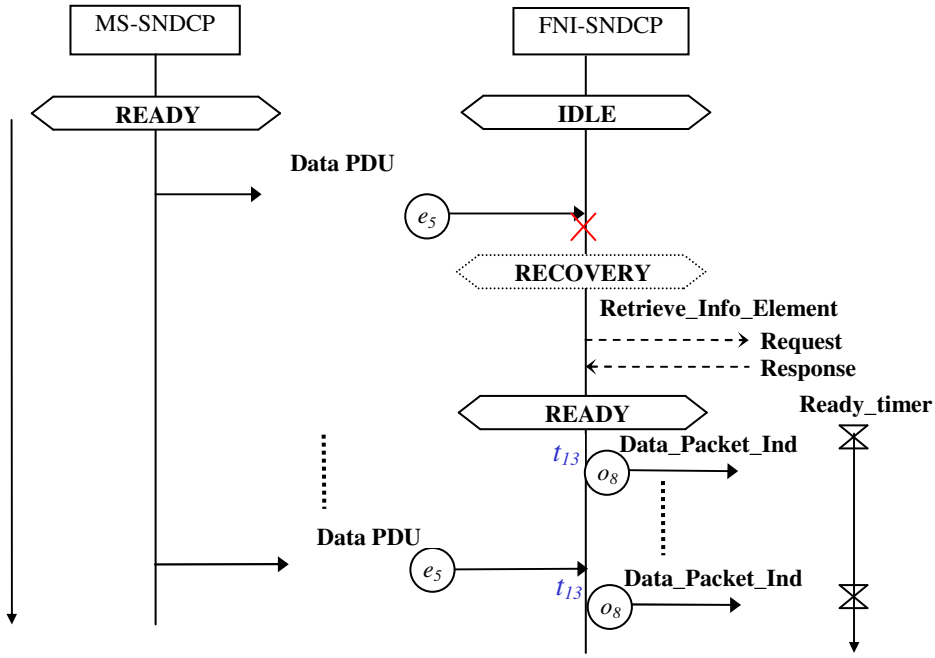

Figure 17: Recovery of an MS initiated communication (Ready)

Figure 17 shows MSC diagram of recovery procedure for an MS initiated communication that is at Ready state. The MS was at Ready state when the FNI SNDCP did crash. The MS SNDCP sends DATA PDU to the FNI SNDCP. The FNI SNDCP receives DATA PDU in an indication from MLE (e_5) at initial state Idle, but since the state-event pair (e_5, s_1) is inconsistent with state transitions T and in addition to the knowledge about the previous failure so it enters Recovery state. The FNI SNDCP entity calls Retrieve_Info_Element to retrieve the saved state and information elements from SIS for the MS in concern. After receiving response from SIS, FNI SNDCP enters Ready state, starts Ready timer, and executes state transition t_{13} where Data_Packet

indication is sent to the upper layer (o_8). The consistency is restored now between FNI SNDCP and MS SNDCP and hence communication can continue as normal.

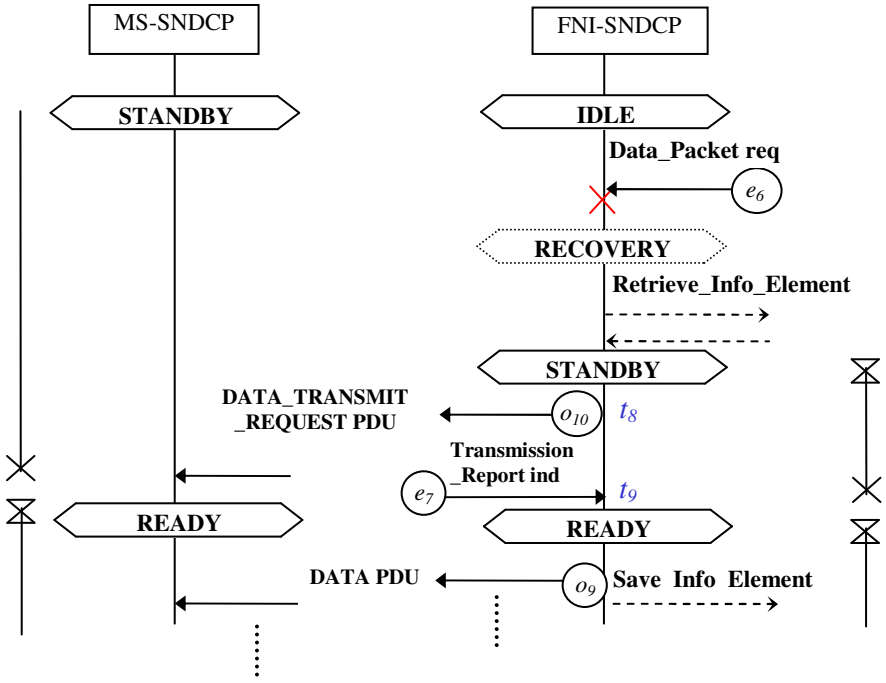

Figure 18: Recovery for an FNI initiated communication (Standby)

Figure 18 shows MSC for recovery procedure for an FNI initiated communication and the MS is currently at Standby state. On the reception of Data_Packet request from the upper layer (e_6) , the FNI SNDCP enters recovery state because the state-event pair (e_6, s_1) is inconsistent with T and then calls Retrieve_Info_Element to retrieve state information (i_2, i_4, i_5, i_6) for the target MS. The FNI SNDCP uses both the retrieved state and time information "*elem_age*" to determine the state of the MS SNDCP, for example the retrieved state may be Ready but the "*elem_age*" value implies that the MS SNDCP must be at Standby (case where MS SNDCP entity had timed out while

the FNI SNDCP entity is down). The FNI SNDCP enters Standby state, starts the Standby timer and executes t_8 (assuming the MS location i_4 is known) where the received packet is queued a_6 and DATA_TRANSMIT_REQUEST PDU is sent in a request to MLE (o_{10}). On the reception of Transmission_Report indication (e_7), the FNI SNDCP entity executes transition t_9 where Standby timer is stopped, Ready state is entered, the queued packet is sent in DATA PDU (o_9) and finally Ready timer is started. The communication between the FNI SNDCP and that MS is now completely recovered and can resume as normal.

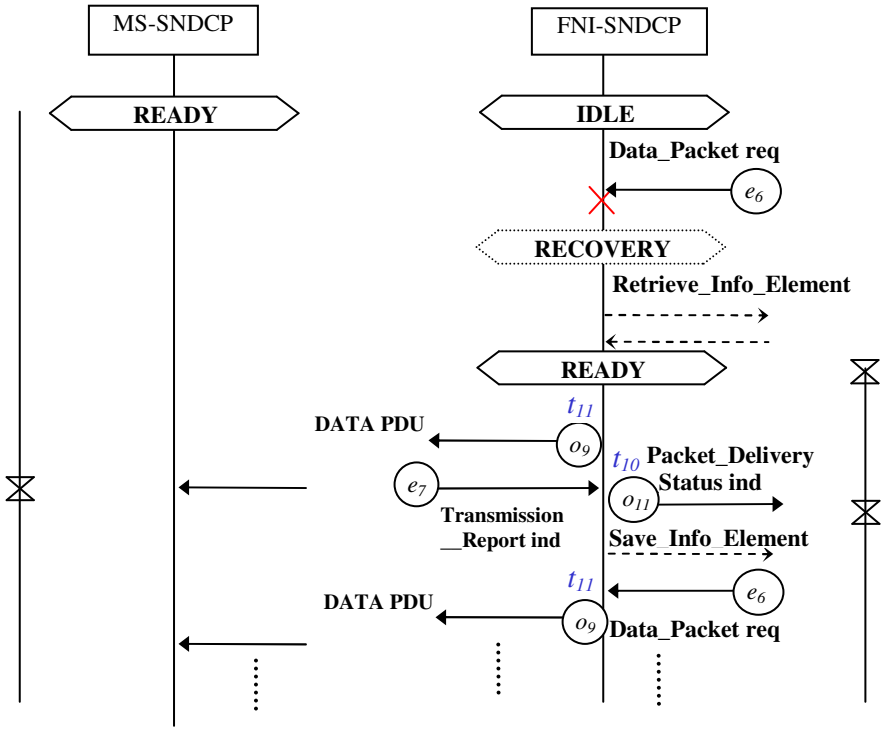

Figure 19: Recovery for an FNI initiated communication (Ready)

The MSC diagram in Figure 19 shows recovery procedure for an FNI initiated communication and the MS is currently at Ready state. On the

reception of Data_Packet request from the upper layer (e_6), the FNI SNDCP enters recovery state because the state-event pair (e_6, s_1) is inconsistent with T and then calls Retrieve_Info_Element to retrieve state information for the target MS. Based on the retrieved information, the FNI SNDCP enters Ready state, starts the Ready timer with value (i_6 - *elem_age* of Ready state) to synchronize with MS Ready timer and then executes t_{11} where the received packet is passed down in a DATA PDU (o_9). On the reception of Transmission_Report indication (e_7), the FNI SNDCP entity executes state transition t_{10} where Packet_Delivery_Status indication is sent (o_{11}) to inform the upper layer about the transfer of the packet and Ready timer is restarted. The MS SNDCP restarts in turn the Ready timer upon receiving the packet. The communication is now recovered with that MS and the next Data_Packet request is processed as normal.

Note that because the state transition based recovery is *event-driven*, that makes it very efficient regarding overhead for two reasons. First, the frequency of information saving during failure-free execution is kept minimal because information is only saved at the correct point of time as mentioned before. Second, the FNI entity only restores the consistency with each peer entity on MS upon the occurrence of first inconsistent state-event pair and not before i.e. any MS entity will be recovered only when there is a communication demand, so as a result the system as a whole will be recovered smoothly and gradually.

Chapter 6

Experimental Testbed and Results

Our aim is to develop a testbed for TETRA packet data where peer-to-peer communication can be generated between protocol entities on MSs and FNI. The STBR method is applied to the FNI entities to study failure recovery in FNI. We use an operational model approach to develop the testbed. By operational model, we mean that the testbed should to a large extent operate as the real system does regarding message exchange and timing, but only provide a restricted form of its functionality. For example, in real system the cell change procedure is started at the MAC layer where signal strength is continuously measured, and when a stronger signal is detected from a neighboring cell, then the MAC layer indicates to the upper layer the cell change. In this example, the signal measurement is not relevant, but the MAC layer should be able to indicate to the upper layer about cell change if that is required by the experiment. The evaluation of the recovery should be done from the end user point of view. Different criteria are of interest to judge the STBR method, first recovery time i.e. the average time to restore pre-failure performance, and second the reliability of the recovery process. Furthermore, the performance overhead incurred by the STBR during free-failure execution is also important.

6.1 Testbed architecture

The testbed consists of four PCs that are connected via Ethernet. The entire testbed architecture is depicted in Figure 20. Each of the machines has an

application and/or a task running on it. We use the word “task” to refer to a software module that is not part of the TETRA packet data standard. The testbed software consists of three applications, namely mobile_station (MS), Base_Station(BS) and Packet Data Server(PDS) and three tasks, namely Mobile_End_User (MEU), State_Info_Saver (SIS) and File_Packet_Sender (FPS). The following items describe each of these applications and tasks individually, explaining their functions and their implementation issues.

1. Mobile_Station (MS): This application provides packet data services to the MS users. It implements SNDCP, MLE, LLC and MAC protocol entities of the TETRA packet data on MS side. The MS application is able to run hundreds of MSs concurrently where each MS runs as a single thread. Each MS thread provides packet data service to one single user. The MS application is implemented in C++ and consists of about 5,000 lines of code. It runs on 1600 MHZ, 512 MB RAM PC with Microsoft Windows 2000.

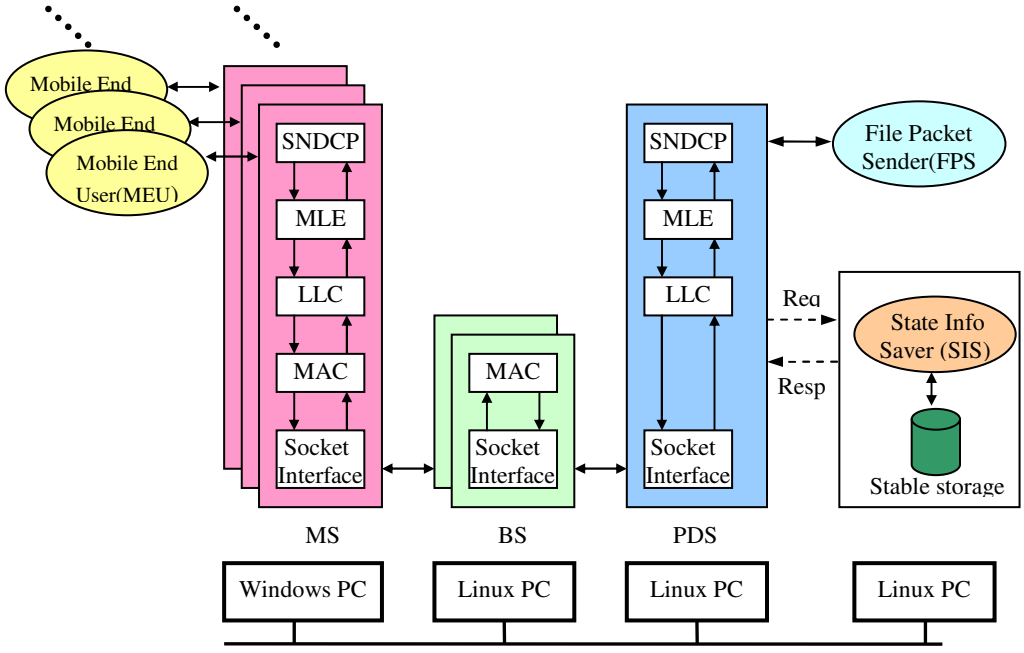

Figure 20: Overall architecture of TETRA packet data testbed

2. Base Station (BS): This application provides main services provided by the BS such as data channel allocation and time slot reservation. Each BS application is assigned a number of channels to serve the mobile stations in its own cell. The data transfer on wireless channels is imitated by data transfer on UDP socket connections. The BS application implements MAC protocol entity and an interface to socket layer for sending and receiving MAC PDUs. Multiple BS applications can be run concurrently on the same machine or different machines where each application is identified by a unique address which is a combination of IP address and socket number. Each BS has a fixed socket that is known for each MS in the cell. This socket acts as the common control channel in real system where any MS that is either at Idle or Standby state utilizes it to initiate communication with the FNI by using random access procedure. The MS that has more data to send/receive will be shortly instructed to move to another packet data channel (another socket) where it can transmit by reserved access. BS uses TDMA access scheme where the channel is divided into timeslots and is shared by multiple MSs, however, each MS may transmit during the timeslots granted by the BS. BS application is written using C and consists of about 2,500 lines of code. It runs on 500 MHZ, 128 MB RAM Linux PC.

3. Packet Data Server (PDS): This application implements the main functionalities of the LLC, MLE and SNDCP protocol entities in the FNI. The PDS application communicates with every BS application and it provides packet data services to all mobile stations created by the MS application. The PDS application sends LLC requests to the MAC entity at BS and receives indications from BS through socket interface. The PDS application has a known socket where any BS can forward the received MAC PDUs from MSs. The PDS in turn sends LLC PDUs in request primitives to the BSs. Each entity of the PDS is implemented as a single thread where inter-communication is done through messages via SAP primitives. The PDS application is written in C

and consists of about 5,000 lines of code. It runs on 548 MHZ, 256 MB RAM Linux PC.

4. *State Information Saver* (SIS): This server task enables FNI entities to easily save their state information according to the recovery protocol developed in section 5.3. The client (FNI entity) needs to link to an SIS library in order to have access to the protocol function calls. The FNI entities communicate with SIS server through sockets. The SIS has a known socket at which the PDS application entities can send the requests defined by the STBR recovery protocol. The SIS server task is implemented in C (about 800 lines of code) and runs on a separate 500 MHZ, 128 MB RAM Linux PC.

5. Mobile End-User (MEU): This is a simple task that uses the packet data services provided by the MS application. There is one MEU task associated with every MS. Each MEU task can request its MS to download files from a destination PC. The MEU tasks has several functionalities such as converting files into packets to be delivered to SNDCP entity for transmission, assembling received packets from SNDCP entity into files, add/verify file checksum, file retransmission, generating different traffic and mobility patterns, collecting experimental statistics, etc. The MEU is written in C++ (about 700 lines of code) runs on the same Windows machine where the MS application runs.

6. File_Packet_Sender (FPS): This is the task that handles the file download requests issued by the MEU tasks. The FPS task owns a number of files of different sizes. The requested file is converted into packets that are sent to the PDS application. The FPS task has a known socket where PDS can deliver messages. The FPS task is implemented in C (about 600 lines of code) and runs on the same machine where SIS runs but it may also run on a separate linux machine.

6.2 Experiment procedure and configuration

Our intention is to test the STBR method on the PDS application. The idea is to force the PDS application to crash and then study its recovery as perceived by the MEU tasks. Therefore, the PDS application is linked to the SIS library to enable PDS entities to register to the SIS and save their state information. Furthermore, the code of the PDS is updated according to the STBR method, where the finite state machine code of SNDCP, MLE and LLC entities are updated to handle Recovery state. Every place in the code where it is necessary to save/retrieve state information is identified and Save_Info_Element/ Retrieve_Info_Element calls are added.

Let us first describe the communication flow that takes place in the testbed. Each MEU task uses its MS to periodically send file download request to FPS task. But before sending any user data, every MS needs to create PDP context activation with the PDS application. If file downloading is not started after 5 seconds, the MEU task retransmits the file download request. Once the FPS task receives the file download request, it converts the requested file into packets and sends them one by one to the PDS SNDCP entity. The FPS task sends the next packet upon receiving Packet_Delivery_Status indication with transfer status “success” from the PDS SNDCP (refer to 5.4.1). The PDS SNDCP entity adds a header to every packet and forwards it as an SDU to the MLE entity which in turn adds a header and forwards it to LLC entity. The LLC entity divides the received SDU into segments (maximum size of 231 bits each) and a header containing SDU number and segment number is added to each segment. The LLC entity then requests the BS application to reserve time slots for all segments in hand. On reception of slots grants from the BS, the LLC entity sends each segment in a timeslot to the BS. The BS application forwards the segments to the appropriate MS. The MS MAC entity receives the segments through the socket and forwards them to the LLC entity. The MS LLC entity assembles the received segments (after removing headers) into a complete SDU

that is forwarded to the MLE entity. The MLE entity removes its header and then forwards the SDU to the SNDCP entity. The MS SNDCP entity extracts the packet and delivers it to the MEU task. The MEU task collects all the packets and reconstructs the file, calculates file checksum, measure download time, etc.

The testbed setup that is used to conduct experiments consists of 2 BSs where each BS is assigned a number of data channels enough to carry the data load in experiments. Each data channel has a gross bit rate of 28 Kbit/sec. Two file sizes are used for experiments 24 and 40 KB, the ideal download time are bout 7 and 11 seconds respectively, while actual time is about twice of that. The number of MSs in each cell and the address information to access BSs can be entered in the MS application interface shown in Figure 21. Cell change rate is set to 10% i.e. about 10% of MSs will move to the other BS every minute.

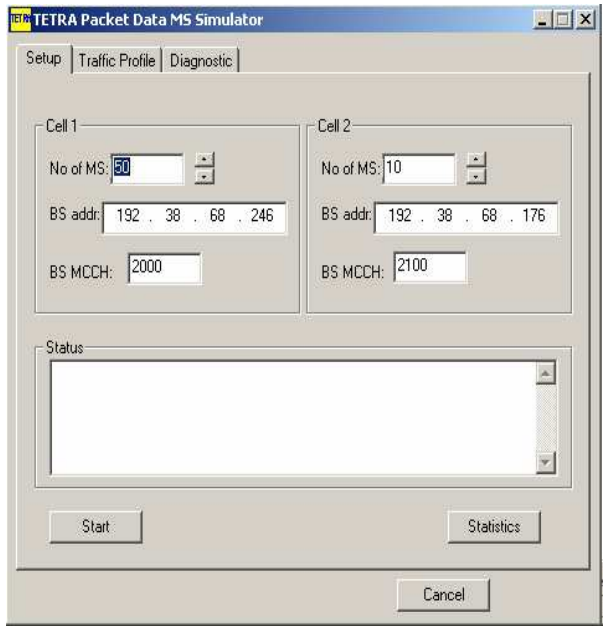

Figure 21: The MS application user interface

Each failure recovery experiment begins by starting all applications and tasks and when the workload reaches steady state(also known as stationary state) i.e.

the average number of packets received by all MEU tasks is relatively stable, a Kill signal is sent at a random instant of time to cause the crash of the PDS application. The PDS application is then restarted after a short time period between 5-15 seconds to compensate for fault detection time. The recovery of the PDS application is then assessed by its effect on MEU tasks both with respect to performance (download time) and correctness (download success).

6.3 Experiments

We conducted 3 sets of experiments. In the first set of experiments there are 50 MSs in both cells and on average one MS is context activated every 1 second. Therefore, it takes about 50*1 seconds to create all PDP context activations i.e. every single MS in the experiment can send and receive data (possibly all together at the same time). Once an MS completes PDP context activation, its associated MEU task issues a file download request within 30 seconds requesting the FPS task to download a 40 KB file size by simply picking a random number between 0 and 30. Upon the reception of all file packets, the MEU task calculates both file checksum to check its integrity and the download time. The MEU task then picks again a random number between 0 and 30 seconds to issue a new file download request. In other words, each MEU task will on average send a new file download request after 15 seconds from finishing the latest file download. The MEU task keeps downloading files until a selected number set by the experiment is reached. The total number of completely downloaded files, number of packets and the average file download time during each time unit is computed for every MEU task. The time unit used in the experiments is 10 seconds. The average file download time is the average of download times of all files that completed downloading in the same time unit. Note that the beginning and the end of file download can not occur in the same time unit unless the download time is less than the time unit.

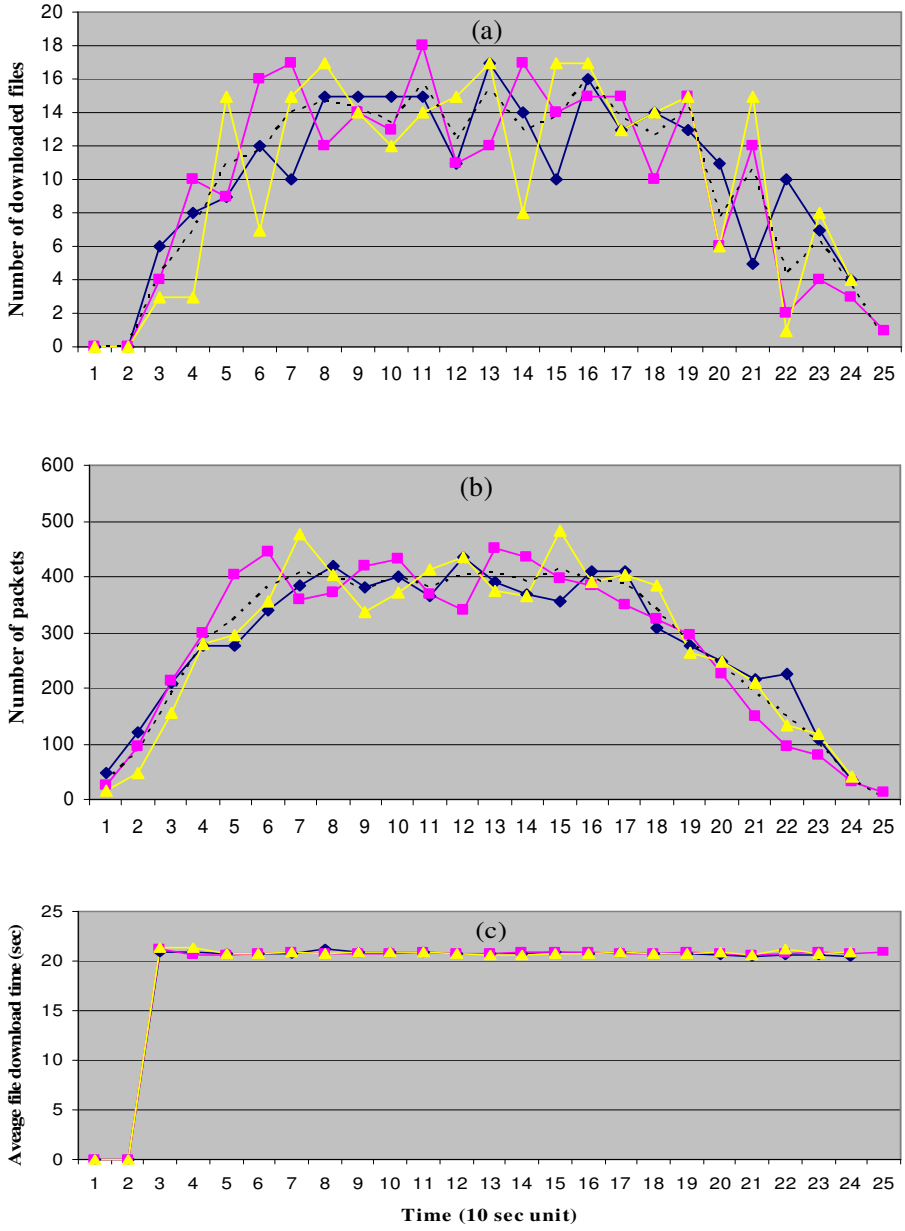

Figure 22: 3 typical failure-free experiments in set #1: (a) Number of downloaded files per time unit for each experiment run; (b) The corresponding number of packets; (c) The average download time of 40 KB file.

Figure 22(a) shows the number of downloaded files for 3 representative experiments in set #1 during failure-free run. A run in which the PDS application does not crash is called a failure-free run. The number of files that finish downloading within the same time unit is found for the entire experiment run and the results are shown as solid curve. As can be seen from the graph, the number of file download completions per unit time is zero at the beginning of each experiment. In the first period of the experiment, the number increases rapidly as more MSs create PDP contexts and thus more MEU tasks start downloading files. In the second period, the number fluctuates around 14 files. In the third and final period, the number declines to zero as more MEU tasks stop downloading because they reached the selected number of files. The dashed curve shows the mean number of downloaded files for all these three experiments. Each MEU task downloads 5 files and then stops so the total number of downloaded files in each experiment in set #1 is 50*5 files. Each experiment takes about 250 seconds from the beginning to the end. Figure 22(b) shows the corresponding number of packets received by all MEU tasks in each time unit for these three experiments. The 40 KB file is equivalent to 28 packets of maximum size of 1500 bytes so the number of downloaded packets in each experiment in set #1 is 50*5*28 packets. The total number of received packets per unit time increases in the first period of the experiment then fluctuates around 400 in the second period and finally declines in the final period. The dashed curve is the mean number of packets across all the three experiments together. The average file download time during these 3 experiments is shown in Figure 22(c). As it can be seen from the figure, it takes about 20 seconds to download the 40 KB file from the FPS task to any MEU task during failure-free run and it is almost constant. As previously mentioned, we intend to induce failure while the workload is at steady state. The steady state as it can be clearly seen from the mean curves in is reached after about 70 seconds from the experiment beginning - when the number of packets is around 400 and

downloaded files is 14 - and sustain about 100 seconds. The steady state load (packets per unit time) can be calculated by using this simple formula:

$$Steady\ load = Number\ of\ MSs \times \frac{file\ size\ in\ packets \times time\ unit}{file\ download\ time + average\ time\ between\ downloads}$$

Thus the expected steady state load in set #1 is about 50 * (28 * 10/ 20+15) = 400 packets per unit time which is equivalent to about 14 files per unit time. The calculated steady state value matches well with the value obtained from the mean (dashed curve) across all the three experiments. It can be also deduced from the calculation that at steady state there will be around 14 MEU tasks downloading simultaneously.

In the second set of experiments there are 100 MSs, and the context activation occurs at a rate of one MS every 0.5 second in order to keep the 50 seconds period to complete all PDP context activations. Each MEU task downloads the 40 KB file 5 times from the PC running the FPS task. Once the MEU task receives the whole file, it picks a random number between 0 and 30 seconds to start the next file download. Figure 23 (a) shows the number of downloaded files per unit time for 3 representative failure-free experimental runs in set #2. As can be seen from the graph, the number of file download completions per unit time increases from 0 to about 30 files in the first period, then fluctuates around 30 files in the second period and finally decreases to 0 in the third and final period. Each experiment takes about 260 seconds and performs 100*5 file downloads. Figure 23 (b) shows the corresponding number of packets received during each time unit of the three experiments. The total number of packets received by all MEU tasks during each experiment in set #2 is 100*5*28, where 28 is the size of the 40 KB file in packets. The average file download time during all these 3 runs is shown in Figure 23 (c). The average file download time is about 20 seconds, exactly the same as in set #1. As can be seen from the

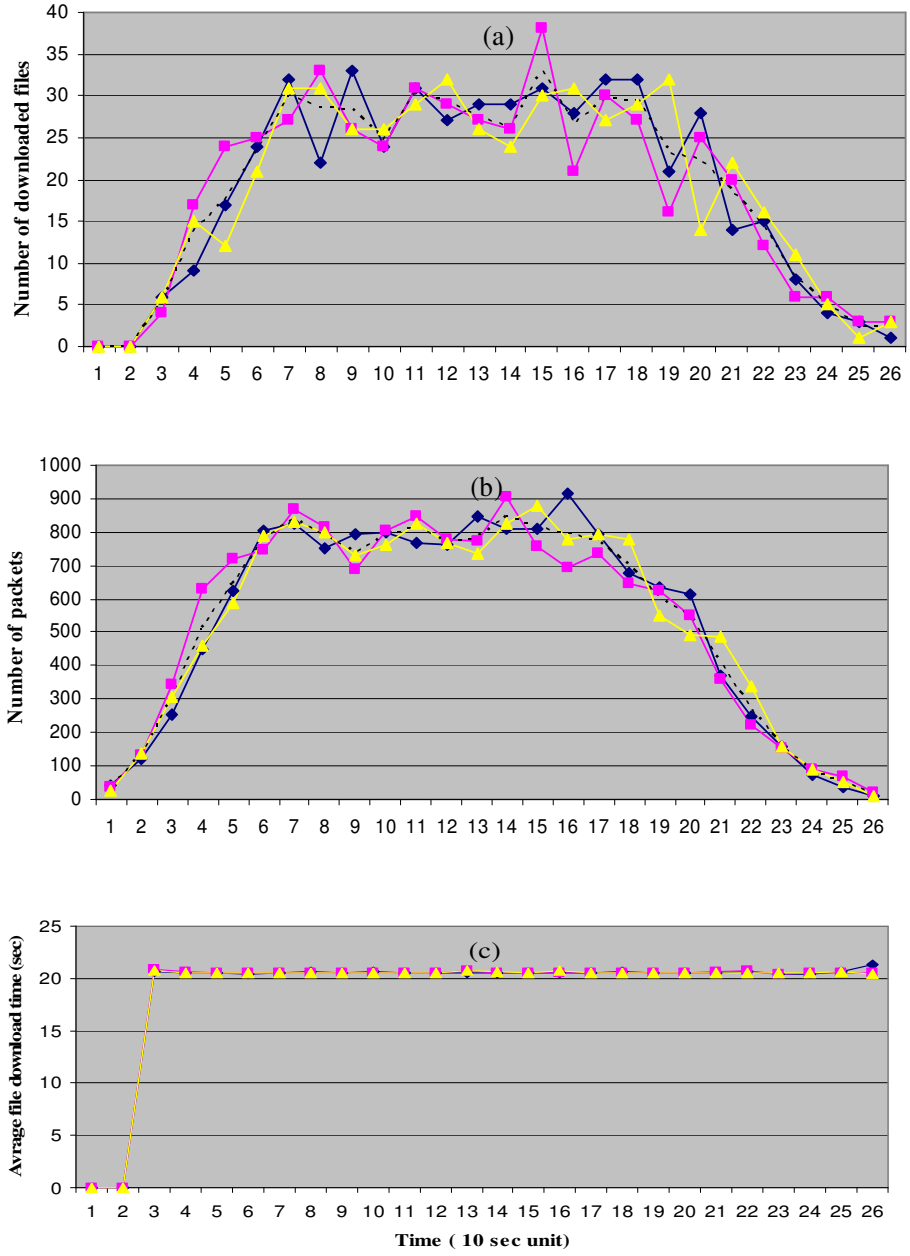

Figure 23: 3 typical failure-free experiments in set #2: (a) Number of downloaded files per time unit for each experiment run; (b) The corresponding number of packets; (c) The average download time of 40 KB file.

mean curve, the steady state is reached between 70 and 170 seconds of the experiment time during which the number of packets per unit time is around 800 and the number of downloaded files is between 25 and 30. The expected steady state load of experiments in set #2 can be calculated as follows:
100 * (28 * 10/ 20+15) = 800 packets per unit time or about 28 files per unit time. That's again match good with the mean curve for the three experiments. Consequently, the number of MEU tasks that will download simultaneously at steady state is about 28. The steady state load in experiments of set #2 is the double of that in set #1 as a result of doubling the number of MSs.

In the third set of experiments there are 200 MSs, and the context activation occurs at a rate of one MS every 0.25 second. Each MEU task downloads a 24 KB file 5 times from the PC running the FPS task. Once the MEU task receives the whole file, it issues a new file download request within 20 seconds by simply picking a random number between 0 and 40 seconds. Figure 24(a) displays the number of the number of downloaded files per unit time for three representative experiments in set #3. In each experiment run, 200*5 files are downloaded from the FPS task during a period of about 260 seconds. The dashed curve plots the mean number of downloaded files across all the three experiments. Figure 24(b) displays the number of received file packets per unit time for the three experiments. The total number of packets which is downloaded during every experiment is 200*5*16, where 16 is the size of the 24 KB file in packets. The dashed curve displays the mean number of packets for these three experiments. Finally, Figure 24(c) shows the average file download time for the three experiments. The average download time for the 24 KB file is about 12 seconds and is relatively constant throughout all the three experiments. The two mean curves for files and packets indicate that that the steady state is reached after 70 seconds i.e. when the number of packets is around 1000 and files around 60, and it lasts about 7 time units before the

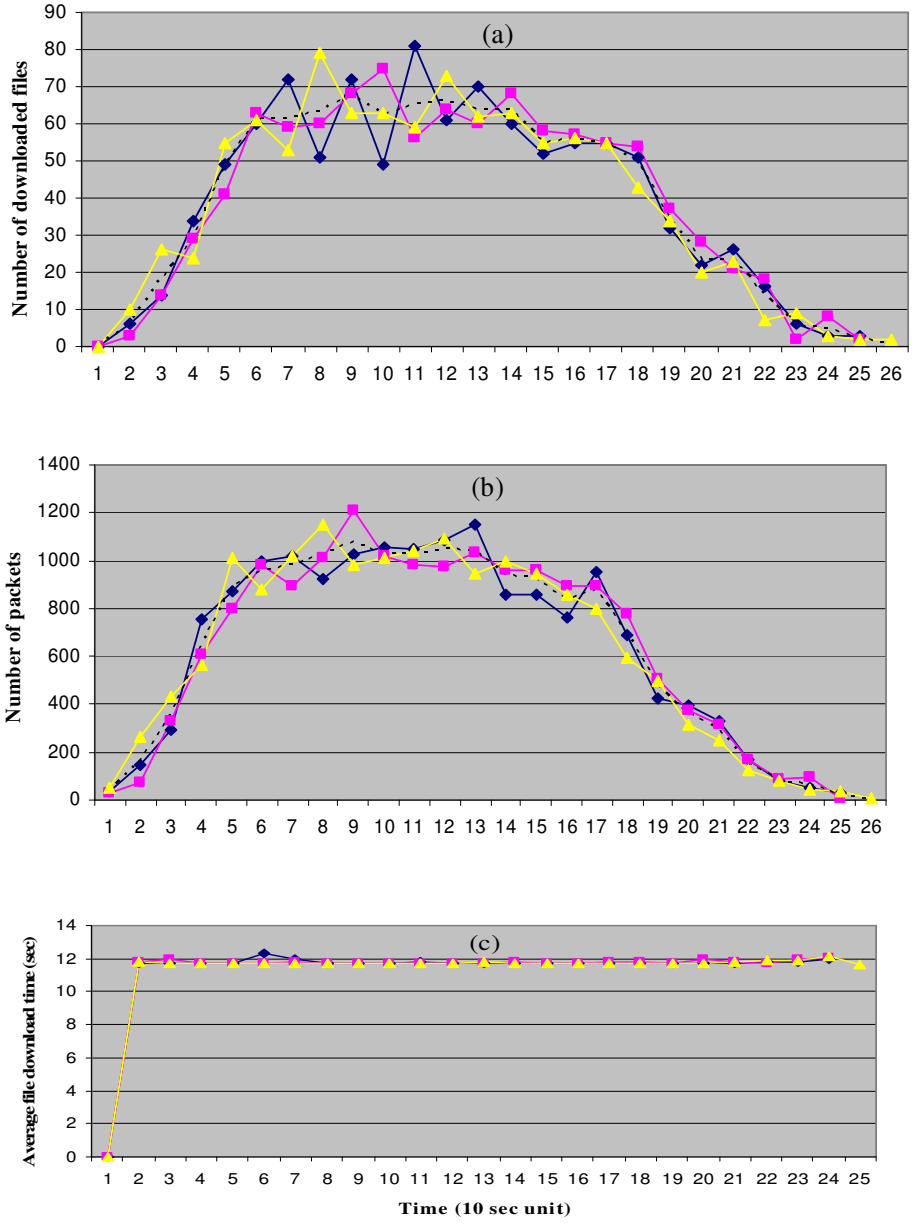

Figure 24: 3 typical failure-free experiments in set #3: (a) Number of downloaded files per time unit for each experiment run; (b) The corresponding number of packets; (c) The average download time of 24 KB file.

curves start declining. The expected number of packets per unit time at steady state load can be calculated as follows:
200 * (16 * 10/ 12+20) = 1000 packets per unit time or about 62 files per unit time. That's again very close to the mean curve for the three representative experiments. Accordingly, the number of MSs that will transmit simultaneously at steady state is going to be around 62. The steady state load in experiment set #2 is about 25% higher than in set #2.

6.3.1 Failure recovery in experiment set #1

To test the failure recovery of the PDS application in the first set of experiments, we intend to force the PDS application to fail by sending a Kill signal while the load is at steady state and then restart it after a short period of time to compensate for the failure detection latency. Figure 25 shows 3 representative experiments where the Kill signal is issued at around 75, 95 and 115 seconds from the experiment beginning respectively and the PDS is restarted after 5 seconds. The instant of the PDS crash may deviate about ± 1 second from the mentioned values. Figure 25 (a) displays the effect of the 5 seconds failure period on the number of files downloaded by the MEU tasks. For the sake of comparison, the dashed curve which plots the mean number of downloaded files for the three previous failure-free experiments in set #1 is added to the graph. The number of file download completions per unit time for all three experiments drops from about 14 to lower than 5 files and then rises sharply to over 20 files right after the PDS restart as can be seen. Correspondingly, the number of received packets per unit time falls from about 400 to around 100 packets and then rise sharply to over 400 after the PDS restart as it can be seen in Figure 25 (b). The sharp rise in the number of files and packets is the result of the fact that the number of MEU tasks that

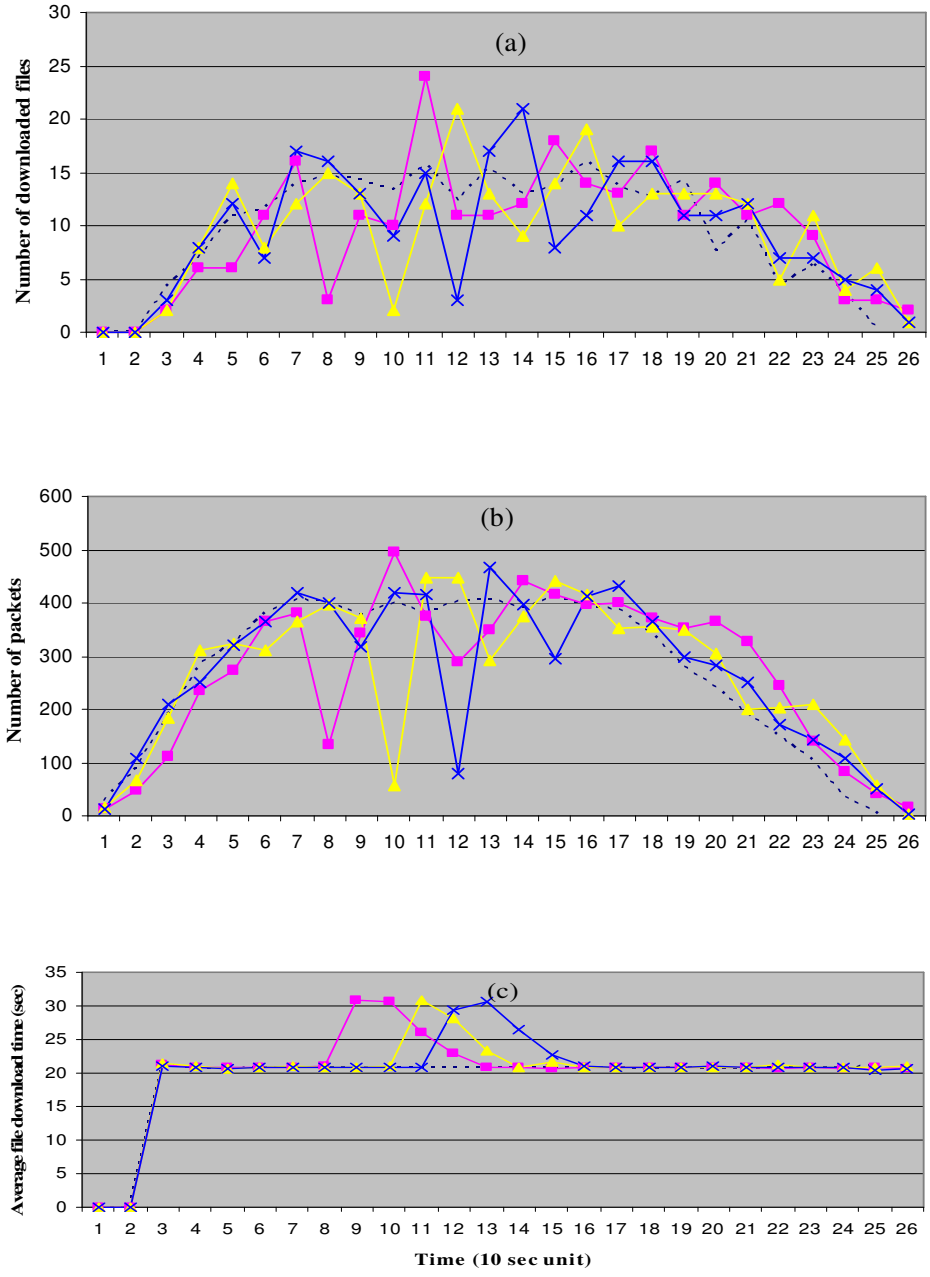

Figure 25: Failure recovery in set #1 for 3 sample experiments where PDS application crashed around 75, 95 and 115 seconds respectively and then restarted after 5 seconds

downloads simultaneously will increase after failure. Every MEU task that was downloading when the PDS crashed besides any new one that tried to download while the PDS is down will retransmit file download request if no file packet is received within 5 seconds. The peak of the first experiment indicates that the number of MEU tasks that downloads simultaneously right after the PDS restart is almost reaching 25, and it exceeds 20 for the second and third experiment. We refer to the experiment killed at 75 second as the first experiment, the experiment killed at 95 second as the second experiment, etc. Figure 25 (c) shows the variation in the average file download time for these 3 experiments. The average file download time is increased from 20 to about 30 seconds after failure and then drops back to 20 seconds within 4 time units from the PDS restart. The average file download can clearly indicate when the impact of failure is removed and the performance is restored.

As can be seen from these experiments, the PDS application eliminates completely the impact of the failure within 40 seconds from its restart. It is also obvious to see that after 4 time units from the sharp rise all three sample experiments tend to re-follow the mean curve nicely. Note that the PDS application starts servicing correctly immediately after its restart but it takes about 40 seconds to eliminate the impact of the failure on the mobile end users. Furthermore, we control the success of the failure recovery in each experiment in set #1 by checking that the total number of downloaded files and packets must be equal to 50*5 files and 50*5*28 packets which correspond to the values obtained under failure-free experiment. Finally we should mention that our procedure of testing and measuring failure recovery is not something that is adopted by the fault tolerance research community nor do we know to any specific procedure but we find our procedure to be very logical and practical.

In the same manner, we conducted experiments where the PDS is restarted after 15 seconds instead of 5. Figure 26 shows 3 typical experiments where the PDS

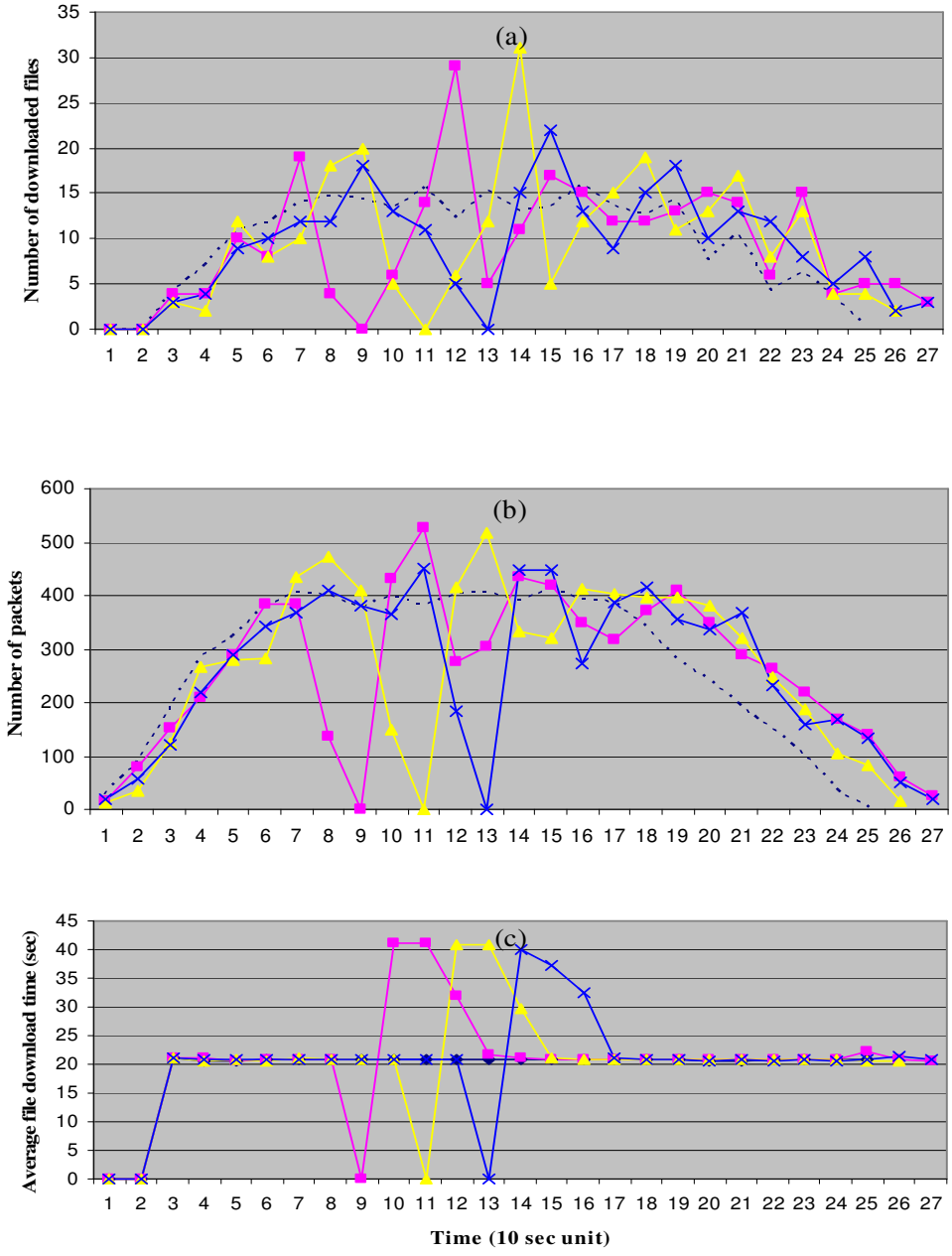

Figure 26: Failure recovery in set #1 for 3 sample experiments where PDS application crashed around 75, 95 and 115 seconds respectively and then restarted after 15 seconds

application is killed at around 75, 95 and 115 seconds respectively and then restarted after 15 seconds. The number of downloaded files drops from around 14 before the PDS crash to zero when the PDS is down for all three experiments as can be seen in Figure 26 (a). The number of files then rises sharply to around 30 files right after PDS restart in the first and second experiment. That also indicates that the number of MEU tasks that are downloading simultaneously is increased to about the double right after the PDS restart. In parallel to the number of downloaded files, the number of packets drops from about 400 to zero during the PDS downtime. The number of packets then rises sharply to around 500 packets to service the increasing number of MEU tasks requesting file downloads. Figure 26(c) displays the average file download time calculated on the MEU tasks during these 3 experiments. The average file download time increases from 20 to 40 seconds. The average file download time peak is increased by 10 seconds (from 30 to 40) relative to the 5 seconds downtime case which match very well with the 10 seconds increase in the PDS downtime. The average file download time is set to zero in the time unit where no file download completion is counted. The PDS application eliminates the impact of failure within 40 seconds from its restart as can be seen. It is clear to see from these sample experiments that the PDS handles the burst of messages coming from MSs after its restart well and within 4 time units it returns back to the steady state level. The experiment takes few more time units than the failure-free run to finish as can be noticed because of the crash and downtime. It is quiet good that the increase in the PDS downtime has no negative impact on the recovery time but the end users will of course observe longer delay.

6.3.2 Failure recovery in experiment set #2

In the second set of experiments, the number of MSs is increased to 100; file size is 40 KB and a new file download is initiated within 30 after the previous file download is finished. Figure 27 shows the experimental data for three representative experiments where the PDS application is killed at around 75, 95 and 115 seconds respectively and then restarted after 5 seconds. As can be seen from Figure 27(a) the average number of downloaded files drops from about 30 to fewer than 10 files per unit time during the PDS crash and then rise to over 40 files right after the PDS restart. The dashed curve displays the mean number of files for the previous three failure-free experiments in set #2. Figure 27(b) shows the corresponding number of packets per unit time for these three experiments. The number of packets drops from about 800 to less than 200 packet per unit time as a result of the PDS crash. The number of packets then rises sharply to about 1000 right after the PDS restart. The average file download time as can be seen in Figure 27(c) increase from 20 seconds before PDS crash to about 30 seconds after PDS restart and then goes back to 20 seconds within 4 time units. That is exactly the same to what happens in experiment set #1 after 5 seconds downtime. As can be seen from these sample experiments in set #2, the PDS is still able to handle recovery quickly and successfully although the workload is increased to the double of that in set #1.

The success of the failure recovery is also controlled for every experiment in set #2 by checking that the total number of downloaded files and packets must be equal to 100*5 files and 100*5*28 packets which correspond to the values obtained under failure-free experiments.

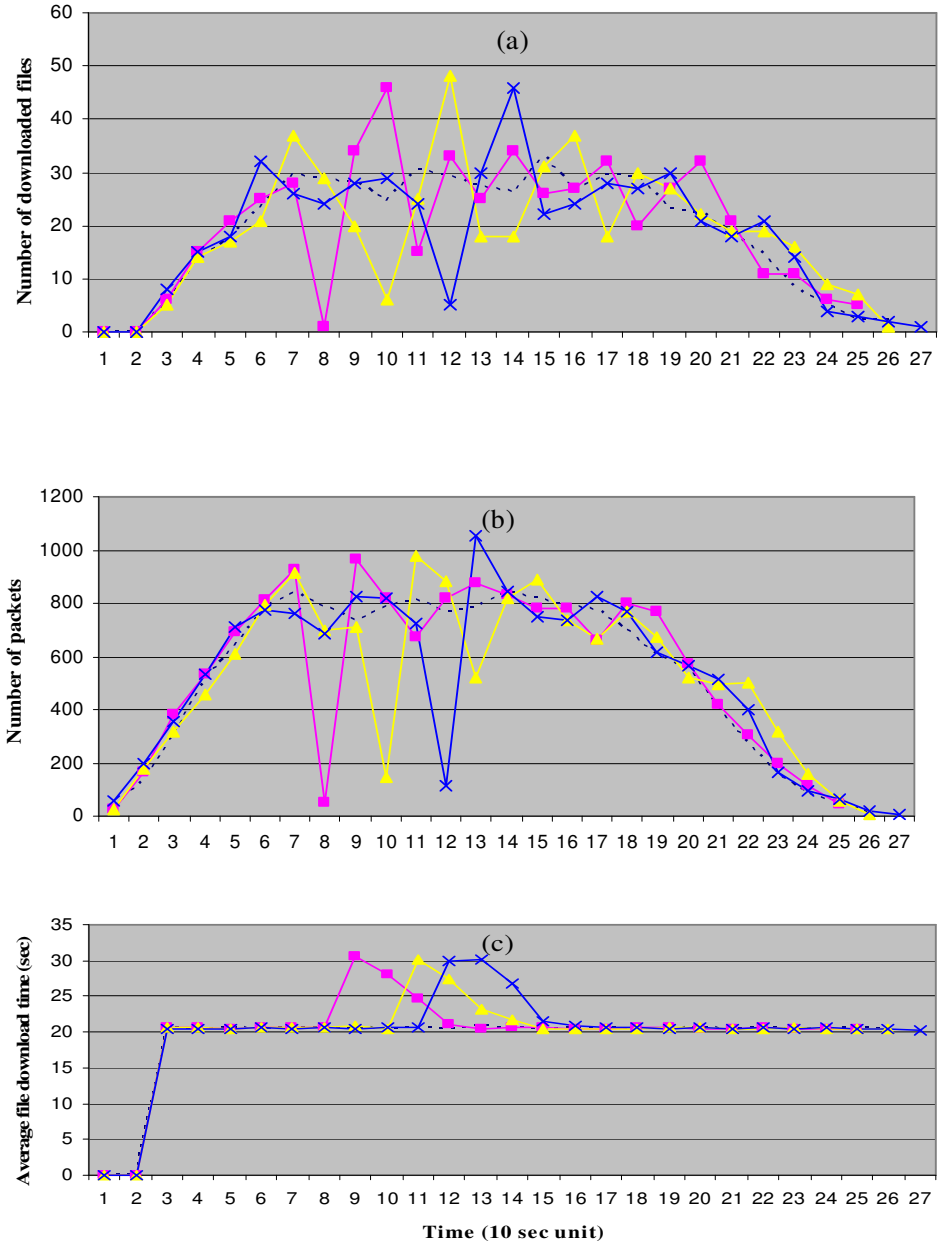

Figure 27: Failure recovery in set #2 for 3 sample experiments where PDS application crashed around 75, 95 and 115 seconds respectively and then restarted after 5 seconds

More experiments were conducted for set #2, but now the PDS downtime is prolonged to 15 seconds. Figure 28 shows three representative experiments in set #2, during which the PDS application is crashed around 75, 95 and 115 seconds and then restarted after 15 seconds. As can be seen in Figure 28(a), the number of downloaded files drops from about 28 before the PDS crash to zero during the PDS downtime. The number of packets then jumps to between 50 and 60 files after the PDS restart. The peaks in the second and third experiment, for example, indicate that there are about 60 MSs that are transmitting simultaneously right after the PDS restart. Figure 28(b) shows the corresponding number of packets per unit time for these three experiments. The number of packets drops from around the 800 packets reached at the steady state to zero and then rise to around 1000 packets right after the PDS restart. The average file download time is zero during the PDS downtime because no file download completion is counted and then increases to 40 seconds right after the PDS start as be seen in Figure 28(c). The average file download time then drops back to 20 seconds within 4 time units. As we can see again, the PDS promptly resumes servicing all MSs that were in the middle of file downloading or requesting new downloads right after its restart and within 40 seconds the impact of the failure is cleared out. That is obvious to see from the average file download time but it can also be seen in the number of files and packets because after 4 time units from PDS restart, each experiment tends to re-follow the course of the mean curve which actually represents the failure-free execution of the PDS.

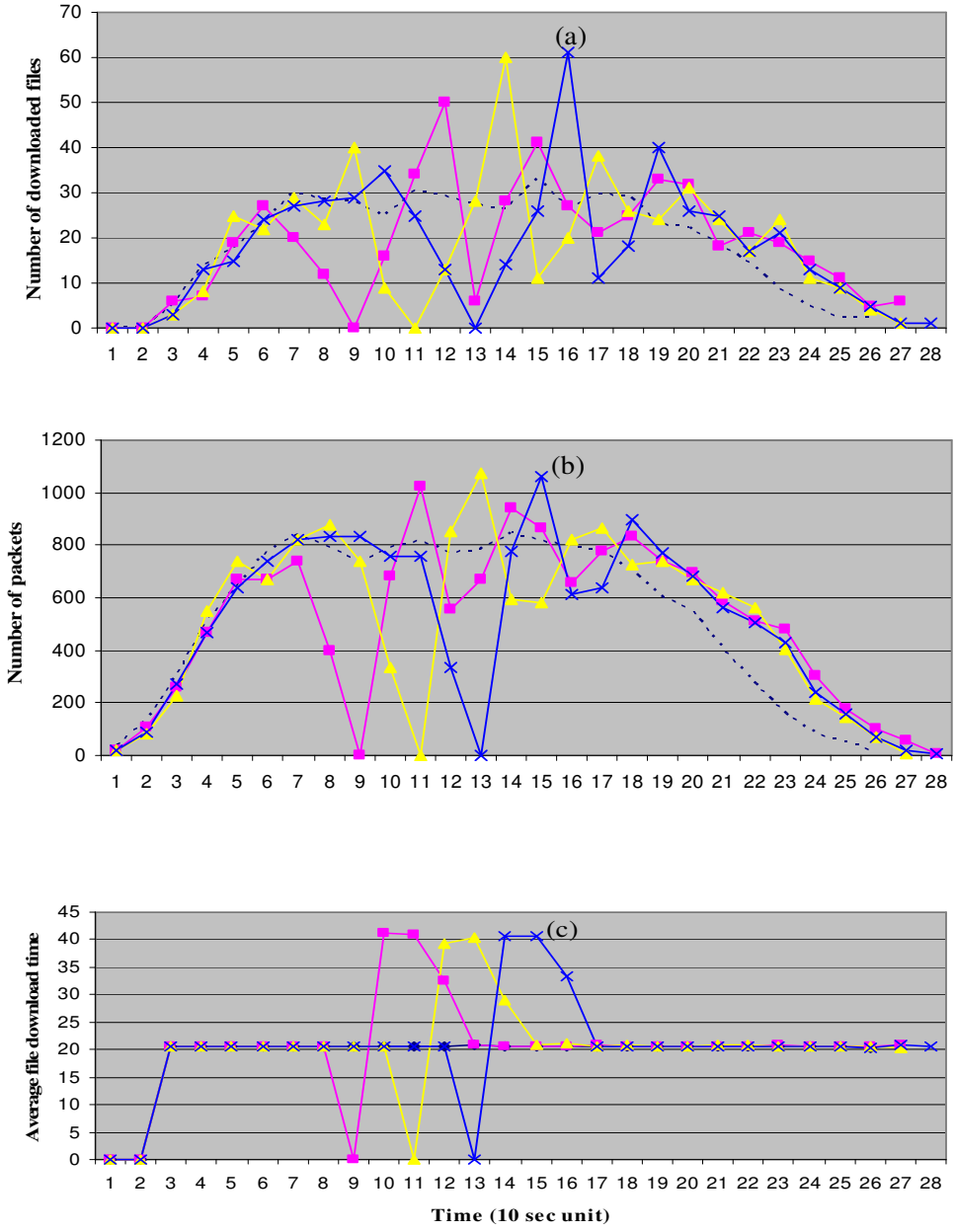

Figure 28: Failure recovery in set #2 for 3 sample experiments where PDS application crashed around 75, 95 and 115 seconds respectively and then restarted after 15 seconds

6.3.3 Failure recovery in experiment set #3

In the third set of experiments, the number of MSs is increased to 200, file size is 24 KB and new file download request is issued within 40 seconds after the previous download finished. Figure 29 shows three typical experiments in set #3 during which the PDS application is forced to crash at around 75, 95 and 115 seconds respectively, and then restarted after 5 seconds. Figure 29(a) shows the number of files that are downloaded by MEU tasks from the FPS task throughout all these experiments. The dashed curve is the mean number of files obtained previously for three failure-free experiments in set #3. As can be seen, the number of downloaded files drops from about 60 files per unit time at steady load to less than 30 during the PDS downtime. The number of downloaded files then jumps to over 100 files right after the restart indicating an increase of the traffic load after the crash. For example, the peak in the first experiment indicates that there are about 120 MSs transmitting simultaneously i.e. about the double of what it is at steady state load. Figure 29 (b) shows the corresponding number of packets received by MEU tasks throughout all these three experiments. The number of packets drops from about 1000 packets per unit time reached at the steady state to fewer than 400 during the PDS downtime and then rise sharply to between 1200 and 1600 right after the PDS restart. Figure 29(c) plots the average file download time throughout all these three experiments. The average file download time as can be seen increases from 12 seconds before the PDS crash to about 22 and then drops back to 12 seconds within 4 time units. The PDS shows again the ability to perform a full recovery within 40 seconds from its restart following a 5 seconds downtime. It is also clear to see that both the number of downloaded files and packets re-follows the mean curve nicely after 4 time units from the PDS restart. The different file size and download rate that is used for experiments in set #3 has not effected the fast and successful recovery. The success of the failure recovery for each experiment is controlled by checking that the total number of

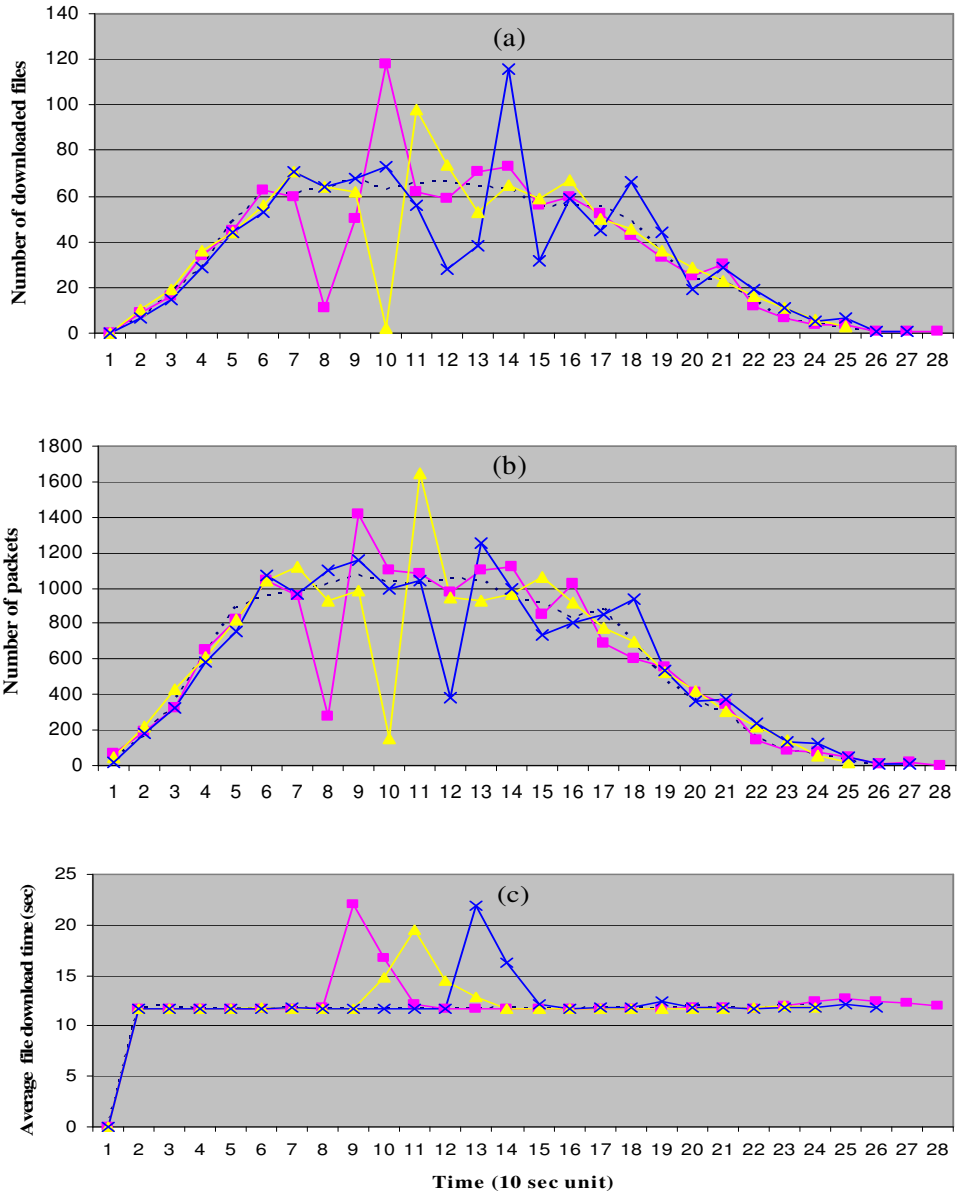

Figure 29: Failure recovery in set #3 for 3 sample experiments where PDS application crashed around 75, 95 and 115 seconds respectively and then restarted after 5 seconds

downloaded files and packets must be equal to 200*5 files and 200*5*16 packets which are equal to the values obtained under failure-free experiment.

Further experiments in set #3 were conducted with the PDS downtime being increased to 15 seconds. Figure 30 shows three typical experiments in set #3, during which the PDS application is crashed around 75, 95 and 115 seconds and then restarted after 15 seconds. As can be seen in Figure 30(a), the number of file download completions per unit time drops from about 60 files reached at steady load before the PDS crash to zero during the PDS downtime. The number of downloaded files then rises sharply to more than 140 as in the first and third experiment which indicates that there are more than 140 MSs transmitting simultaneously right after the PDS restart. The number of packet received per unit time by the MEU tasks during the course of these experiments is shown in Figure 30(b). The number of packets drops from about 1000 packet per unit time before the PDS crash to zero during the PDS down time. After the PDS restart, the number of packets jumps to around 1600 packet per unit time as can be seen in these three experiments. The average download for the 24 KB file is increased from 12 seconds before the PDS crash to about 32 seconds after the PDS restart as can be seen in Figure 30(c). The average file download time then falls down to 12 seconds within 40 seconds from the PDS restart. As we can see again the impact of the failure is cleared out by the PDS after 4 time units from its restart. It is also to clear to see that packet and downloaded file curves for all three experiments re-follows very nicely the course of their respective mean curve after 4 time units the PDS restart. The experiment duration, however, will take about two more time units to finish than the failure-free run as can be noticed.

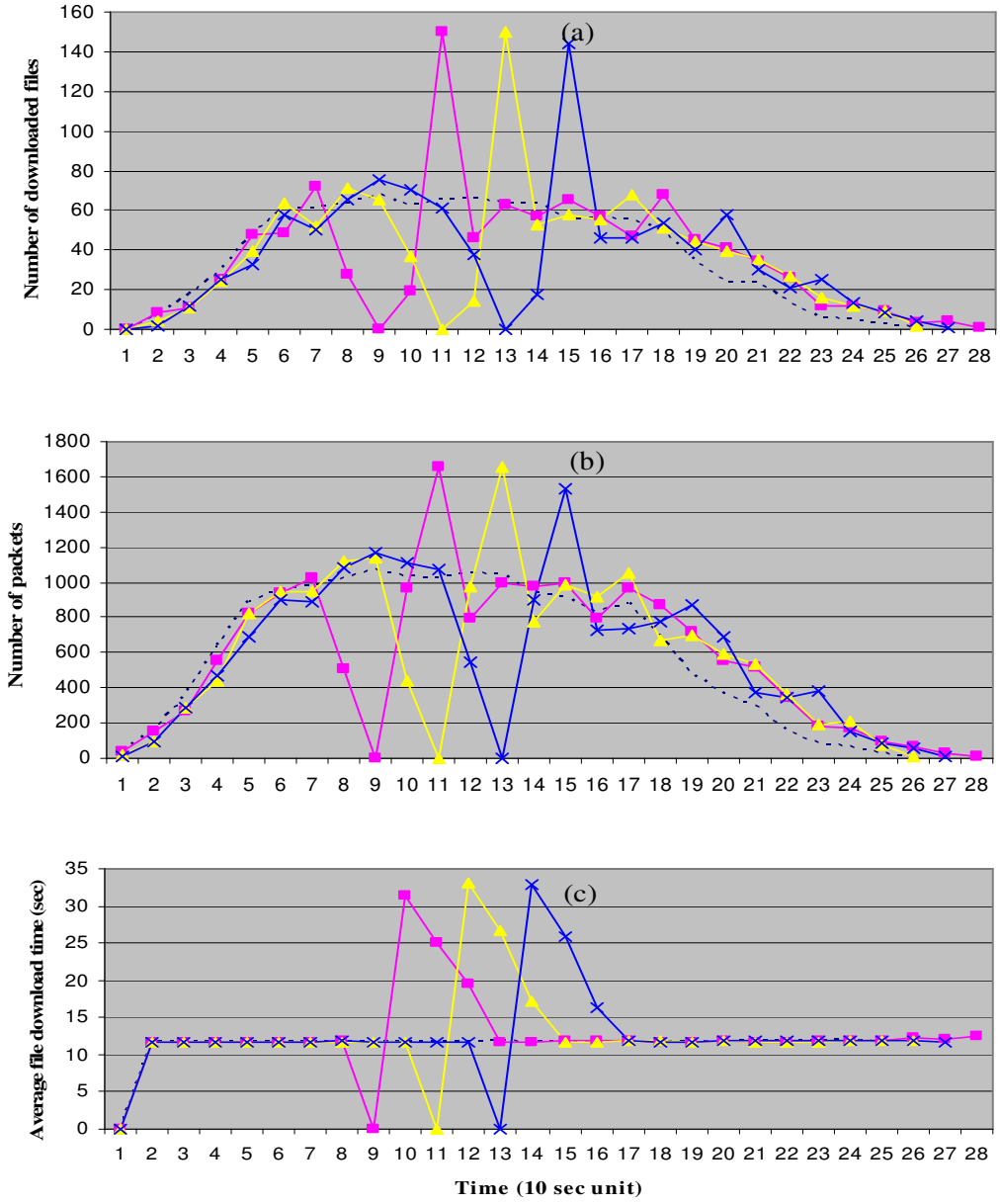

Figure 30: Failure recovery in set #3 for 3 sample experiments where PDS application crashed around 75, 95 and 115 seconds respectively and then restarted after 15 seconds

6.3.4 Failure-free overhead

We used the Linux "top" command to measure the overhead that the STBR method caused on PDS application during failure free execution i.e. the cost of saving and updating the PDS state information on the SIS. To do that, the CPU usage of the PDS application is read by "top" at an interval of 10 seconds during failure-free run for a number of experiments in every experiment set. We then replaced the PDS application with the original one that is not updated with STBR and repeated the same procedure. Figure 31 shows two experiments in set #3, in the first experiment the PDS runs with STBR but in the second experiment STBR is not used. As can be seen, the first experiment uses the CPU slightly more than the second one. The highest CPU usage in the first experiment is 4.7% and 4.5% in the second so the overhead is about 4.4 % in this example. The accuracy in the decimal portion of the results may be not good but this is not very significant. During all conducted experiments, the overhead never exceeded 5% in all experiment sets. This is a low overhead as we previously expected due to the non-blocking socket mode and client/server model used by the STBR.

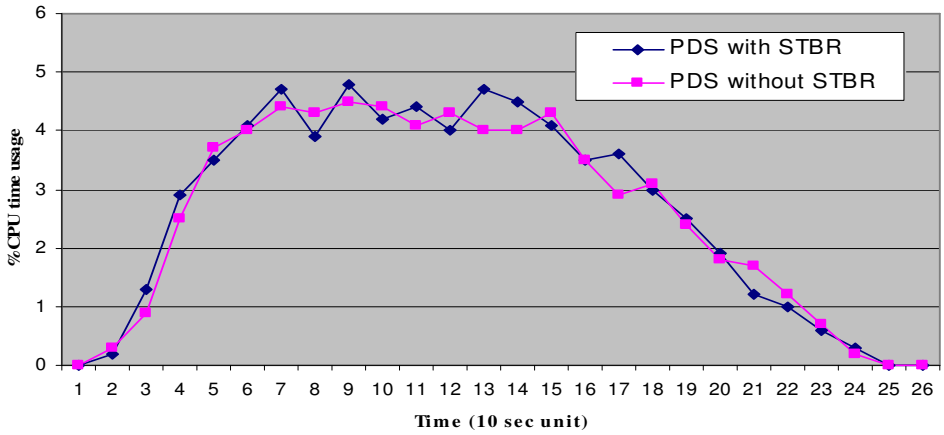

Figure 31: The PDS CPU time usage of two experiments in set #3. The PDS in the first experiment is updated with STBR method but not in the second.

6.4 Experiments summary

Table 6-1 shows a summary of the three experimental sets and the achieved results.

As far as we can see from these results and hundreds of other experiments that we conducted, the STBR method proofed to be able to achieve fast and successful recovery. In each experiment of these three sets and no matter if the PDS downtime is 5 or 15 seconds, the PDS could promptly resume service after its restart and 40 seconds later the impact of the crash failure disappeared. Furthermore the failure-free overhead found to be less than 5% for all experiments where the CPU usage is measured and there were no apparent difference between sets. There were cases where the PDS application did crash during the recovery but after every investigation and hard debugging work it always turned up to be an implementation error somewhere in the testbed software and had nothing to do with the STBR method.

Experiments	Set #1		Set #2		Set #3	
Number of MSs	50		100		200	
File size (KB)	40		40		24	
Time to next file download (sec)	random[0,30]		random[0,30]		random[0,40]	
Average file download time (sec)	20		20		12	
Failure downtime (sec)	5	15	5	15	5	15
Highest average file download time during failure recovery (sec)	30	40	30	40	22	32
Recovery time (time units)	4	4	4	4	4	4

Table 6-1: Summary of the failure recovery experiments

Chapter 7
Conclusions and Discussion

7.1 Conclusions

The use of mobile data services is increasing rapidly due to its huge potential to offer effective solutions to various tasks in many sectors, e.g. public safety, transportation, public health, etc. Consequently, the demand for highly reliable and available data services is expected to be a top priority as users rely more and more on data services to perform their work, especially in the security and public safety fields.

The goal of our research was to develop a failure recovery method that can be used to achieve high availability in mobile data communications systems. We reviewed the best known recovery methods in fault tolerance research and pointed out their general limitations. We then analyzed the characteristics of mobile data communications systems and based on that a number of requirements that need to be fulfilled by any recovery method are determined. Our major contribution in this thesis is the development of the State Transition based Recovery (STBR) method. The STBR is a novel failure recovery method that is based on behavioral model of the communication protocols. The behavior of the communication protocol is modeled by our adapted Communicating Extended Finite State Machine (CEFSM). We took TETRA packet data as a concrete case study and implemented an experimental

testbed to generate the communication traffic between mobile stations and the fixed network infrastructure in accordance to the TETRA standard.

The results of the experiments we conducted on the testbed were very encouraging. The PDS application of the infrastructure which is updated with STBR could instantly resume servicing mobile users when it restarts after its crash as if no failure had happened. Furthermore, the PDS application was able to clear out the impact of its failure on mobile users in less than a minute from its restart. In addition, the performance overhead caused by the STBR during failure-free execution was experimentally found to be less than 5%. Based on these promising results, we belief that high availability (five nines) can be technically achieved in mobile data communication by using the STBR method combined with a quick failure detection mechanism. The high availability can still be maintained even in the presence of permanent faults in the code as long as they are not activated too often as it is normally the case for reliable software.

We can not directly compare our work with other works for many reasons. For example it is not useful to compare it with rollback or replication based recovery methods because they do not work if the fault is permanent. In case of N-version, we would need to implement at least 3 versions of the PDS application by different teams and then forward messages coming from BS application to all versions, and moreover find a way to vote between messages issued from all PDS versions. This is a huge task which requires both resources and time knowing that it took us no less than 200 workdays to implement the PDS application alone. Even then, we suspect that the N-version or other failure recovery methods can meet the real time requirements. We actually do not know any comparative study between the different existing methods. This is probably a shortage from a scientific point of view, but we believe that it will be technically difficult to compare these different methods on experimental level. Finally, in order to compare the STBR with a customized solution that is used by a commercial communication system, we need first to implement STBR in

that system then try to compare these two solutions. That could be a goal for a future project.

We have learned many lessons during the many thousands of hours we used in designing and implementing the applications for the TETRA testbed, then adding STBR, and finally testing and fixing the problems. Based on our previous experience in developing real time embedded communication systems, we estimate that these lessons can be very valuable for communication software development. We discovered during testing the recovery of the PDS application that many bugs and unforeseen scenarios are more likely to show up because the application is exposed to any event at any instant of time. The root causes of these problems are different e.g. design, coding, misinterpretation, or lack of the specifications in the standard. The specifications in any communication standard can not count for all possible scenarios that can occur in the real system and is not supposed to do that. The last and final details fall on the shoulders of the software developers to treat them. Applying STBR thus can aid in performing very thorough test and consequently save a lot of problems before they may be discovered later in the field. We also found out that the specifications of the communications standards (at least TETRA) are flexible enough to contain the STBR method. We have not met with any scenario that was impossible to recover or required a protocol modification. We believe that this will be the case for every well designed protocol.

7.2 Pros and cons

In this section, we try to discuss the benefits, disadvantages and opportunities of using the STBR method as we see it.

The STBR method is a white box approach that requires knowledge about the application to be recovered. The main disadvantage of choosing this approach is that the burden of handling recovery is placed on the system designer and application programmer. However, because the model based

STBR method uses the white box approach it can save the minimum amount of information at the lowest frequency and thus a minimum performance overhead with lowest impact on real-time communication.

The cost of applying STBR to the communication systems is expected to be between low and middle. There are different reasons to support this expectation. As a rule of thumb, the cost to design, implement and test a protocol entity is proportional to the size of its finite state machine determined by the number of states and input events. Extending state transition diagram with one extra "Recovery" state will imply more designs scenarios, more code and more test cases, but this increase will not become significantly high. With respect to the cost of hardware, it is also moderate due to the fact that STBR adopts 2N and N+1 redundancy.

Furthermore, the autonomous recovery principle gives the flexibility to select only the most critical entities or nodes and add STBR capabilities to them.

The ability to recover quickly and reliably from failures will not only improve the availability of the system but it will also make it easier for failure detection technique to judge if the system is operational or not. The failure detection mechanism may early initiate failure recovery procedure by relying on the fast and successful recovery of the STBR.

The STBR method opens the opportunity for "hot update" technology by analogy with "hot swap" for hardware. In other words, it may be possible to replace the running infrastructure application with a newer version that immediately continues to service the mobile users in the field after retrieving their state information from the SIS.

Finally, the STBR method has one more strong advantage which is portability as the method does not depend on any type of hardware or operating system. This gives the designers the free choice to choose the hardware and software that meet their requirements. This is not the case for the rollback and

replication based methods because the accompanied middleware is usually written for a specific operating system and hardware.

7.3 Discussion

Should we live with system down time in the range of hours per year? Or try to develop systems with downtime in the range of minutes? Should we stick to the traditional way of designing and implementing communication systems? Or should we take the additional step and the associated risks? This is a difficult decision to make and this decision probably concerns system designers and developers more than project managers. But there is always a risk by introducing new technology.

Based on our experience, we believe that the most effective approach to develop communication systems is to select the best methods from the beginning. Trying to have a system up and running very quickly to reduce time-to-market and then finally fix the issues of availability and reliability may not lead to the shortest time in the long run. The consequences can be high development and maintenance costs, damaged reputation, and a final system filled with patches with no way to the optimal solution.

The problem here is that it is impossible to foresee the number problems and their size ahead, and when the system is declared to be ready for operation how one can determine if the developed system could had been better both with respect to quality (availability and reliability) and cost. To do that, we need to re-build the system with different approach and methods and then do the comparison. That is not realistic, and the common answer is that there is always place for improvement.

Finally, what is the chance for adopting the STBR in a commercial data communications system? This is an interesting question that we frankly can not give a direct answer to. But the best candidate would be a new started project

that has availability and reliability as top priorities and ready to follow the key principles of building high available system i.e. redundancy, modularity, detection and failure recovery. There is no doubt that customers need high availability and reliability communication systems. However, while no such system is yet in place, customers can not exert hard pressure on system suppliers but have to live with what exists.

Appendix A. SNDCP PDU formats

This appendix lists the PDUs of the TETRA SNDCP protocol and their contents [ETSI03].

Table A-1: SN-ACTIVATE PDP CONTEXT DEMAND PDU

Field name	Length(bits)
SN PDU type	4
SNDCP version	4
NSAPI	4
Address type identifier in demand	3
IP Address IPv4	32
Packet data MS Type	4
PCOMP negotiation	8
Number of Van Jacobson compression state slots	8
Number of compression state slots, TCP	8
Number of compression state slots, non-TCP	16
Maximum interval between full headers	8
Maximum time interval between full headers	8
Largest header size in octets that may be compressed	8
Access point name index	16
DCOMP negotiation	varies
Protocol configuration options	varies

Table A-2: SN-ACTIVATE PDP CONTEXT ACCEPT PDU

Field name	Length(bits)
SN PDU type	4
NSAPI	4
PDU priority max	3
READY timer	4
STANDBY timer	4
RESPONSE_WAIT timer	4
Type identifier in accept	3
IP Address IPv4	32
PCOMP negotiation	8
Number of Van Jacobson compression state slots	8
Number of compression state slots, TCP	8
Number of compression state slots, non-TCP	16
Maximum interval between full headers	8
Maximum time interval between full headers	8
Largest header size in octets that may be compressed	8
Maximum transmission unit	3
SNDCP network endpoint identifier	16
FNI IPv6 information	98
FNI Mobile IPv4 information	71
DCOMP negotiation	varies
Protocol configuration options	varies

Table A-3: SN-ACTIVATE PDP CONTEXT REJECT PDU

Field name	Length(bits)
SN PDU type	4
NSAPI	4

Activation reject cause	8
Protocol configuration options	varies

Table A-4: SN-DATA PDU

Field name	Length(bits)
SN PDU type	4
NSAPI	4
PCOMP	4
DCOMP	4
N-PDU	varies

Table A-5: SN-DATA TRANSMIT REQUEST PDU

Field name	Length(bits)
SN PDU type	4
NSAPI	4
Logical link status	1
Enhanced service	1
Resource request	varies
SNDCP network endpoint identifier	16
Reserved	20

Table A-6: SN-DATA TRANSMIT RESPONSE PDU

Field name	Length(bits)
SN PDU type	4
NSAPI	4

Accept/Reject	1
Transmit response reject cause	8
SNDCP network endpoint identifier	16

Table A-7: SN-DEACTIVATE PDP CONTEXT DEMAND

Field name	Length(bits)
SN PDU type	4
Deactivation type	8
NSAPI	4
SNDCP network endpoint identifier	16
Reserved	12

Table A-8: SN-DEACTIVATE PDP CONTEXT ACCEPT PDU

Field name	Length(bits)
SN PDU type	4
Deactivation type	8
NSAPI	4
SNDCP network endpoint identifier	16
Reserved	11

Table A-9: SN-PAGE REQUEST PDU

Field name	Length(bits)
SN PDU type	4
NSAPI	4
Reply requested	1
SNDCP network endpoint identifier	16

Table A-10: SN-RECONNECT PDU

SN PDU type	4
Data to send	1
NSAPI	4
Enhanced service	1
Resource request	variable
SNDCP network endpoint identifier	16
Reserved	19

Table A-11: SN-END OF DATA

Field name	**Length(bits)**
SN PDU type	4
Immediate service change	1
Reserved	4

Bibliography

[Acharya94] A. Acharya and B. Badrinath, Checkpointing Distributed Applications on Mobile Computers, Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems, pp. 73- 80, Sep 1994.

[Avizienis77] A. Avizienis and L. Chen, On the implementation of N-version programming for software fault tolerance during execution, Proceedings of the IEEE COMPSAC 77, pages 149–155, Nov 1977.

[Birman94] K.P. Birman and R. van Renesse, Reliable Distributed Computing with the Isis Toolkit, IEEE Computer Society Press, Mar 1994

[Birman96] Kenneth P. Birman, Building Secure and Reliable Network Applications, chapter 1 fundamentals page 35, Manning publishing company and Prentice Hall, 1997.

[Budhiraja93] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. Distributed Systems, chapter8: The Primary-Backup Approach, pp 199–216. 2nd edition, 1993.

[Byun02] Young Joon Byun, Beverly A. Sanders and KiSook Chung, A Pattern Language for Communication Protocols, Proceedings of the 9th Pattern Languages of Programming Workshop (PLoP), 2002

[Campos95] Reinaldo V. Campos, Edmundo de Souza e Silva, Availability and Performance Evaluation of Database Systems under Periodic Checkpoints, pp. 269-277, FTCS 1995

[Casanova97] H. Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems, The International Journal of Supercomputer Applications and High Performance Computing, 11(3):212–223, 1997.

[Chandra98] S. Chandra, Peter M. Chen, How Fail-Stop are Faulty programs ?, 28th International Symposium on Fault-Tolerant Computing, pp 240-249, June 1998.

[Chandra00a] Subhachandra Chandra, An Evaluation of the Recovery-Related Properties of Software Faults, PhD thesis, University of Michigan, Sep 2000.

[Chandra00b] S. Chandra and Peter M. Chen, Whither Generic Recovery from Application Faults? A Fault Study Using Open-Source Software, Proc. International Conference on Dependable Systems (DSN 2000), June 2000

[Chandy85] Chandy M. , Lamport L. ,Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 31*, 1, 63–75., 1985

[Cristian91] Cristian F., Jahanian F. , A timestampbased checkpointing protocol for long-lived distributed computations, In Proceedings, Tenth Symposium on Reliable Distributed Systems, pp.12–20, 1991.

[Cukier98] M. Cukier et al., AQUA: An Adaptive Architecture that Provides Dependable Distributed Objects , Proceeding IEEE Symposium on Reliable Distributed Systems (SRDS-17), pp. 245-253, 1998

[Elnozahy96] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang, A Survey of Rollback-Recovery Protocols in Message-Passing Systems, Technical Report CMU TR 96-181, Carnegie Mellon University, 1996

[ETSI03] ETSI, Terrestrial Trunked Radio (TETRA); Voice Plus Data (V+D); Part 2: Air Interface (AI), EN 300 392-2 V2.4.1, Oct 2003

[Gray86] Jim Gray. Why do computers stop and what can be done about it?, In Proceedings of the 1986 Symposium on Reliability in Distributed Software and Database Systems, pages 3–12, Jan 1986.

[Gray91] Jim Gray and Daniel P. Siewiorek, High-Availability Computer Systems, IEEE Computer 24, pp. 39–48, Sep 1991.

[Han99] S. Han and K. G. Shin, Experimental evaluation of behavior-based failure-detection schemes in real-time communication networks, IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No 6,pp 613-626, June 1999.

[ITU94] ITU-T, Information Technology–Open Systems Interconnection–Basic reference model: The basic model, ITU-T Recommendation X.200, July 1994.

[ITU96] ITU-T, Recommendation Z.100: Specification and Description Language (SDL), ITU-T, Geneva, 1996.

[Lowell00] David E. Lowell, Subhachandra Chandra, Peter M. Chen, Exploring Failure Transparency and the Limits of Generic Recovery, Proceedings of the 2000 Symposium on Operating Systems Design and Implementation (OSDI), pp. 289-304, Oct 2000

[Mahmood88] Mahmood, A., and E.J. McCluskey, Concurrent Error Detection Using

Watchdog Processors- A Survey, *IEEE Trans. on Computers*, Vol. 37, No. 2, pp. 160-174, Feb 1988.

[OMG02] Object Management Group (OMG), Unified Modeling Language (UML), version 1.4 OMG Standard, Nov 2002.

[Pradhan95] Dhiraj K. Pradhan, Fault-Tolerant Computer System Design, ISBN 0-13-057887-8, Prentice-Hall, 1996.

[Pradhan96] Dhiraj K. Pradhan, P. Krishna, and N.H. Vaidya, Recoverable Mobile Environment: Design and Trade-off Analysis, Proceedings of the 26th
International Symposium on Fault Tolerant Computing, pp. 16-25, June 1996.

[Randell75] B. Randell , System structure for software fault tolerance, IEEE Transaction Software Engineering, vol. SE-1, pp. 220–232, June 1975

[Schneider84] Fred B. Schneider. Byzantine Generals in Action: Implementing Fail-Stop Processors. ACM Transactions on Computer Systems, 2(2):145–154, May 1984

[Schneider93] Fred B. Schneider, Distributed Systems, chapter 7: Replication Management using the State-Machine Approach, pp 169–197, 2nd edition, 1993.

[Storm85] Strom R., Yemini S., Optimistic recovery in distributed systems. ACM Transactions on Computer Systems, Vol. 3, No. 3, pp 204-226, Aug 1985.

[Traverse88] P. Traverse, AIRBUS and ATR system architecture and specification, editor, Software Diversity in Computerized Control Systems. Springer Verlag, pp. 95–104, 1988

[Wensley72] J. H. Wensley, SIFT Software Implemented Fault Tolerance, FJCC,
pp. 243-253, 1972

[Williams83] J. F. Williams, L. J. Yount, and J. B. Flannigan, Advanced autopilot flight director system computer architecture for Boeing 737-300 aircraft, In AIAA/IEEE 5th Digital Avionics Systems Conference, Seattle, WA, Nov 1983.

[Yao99] B. Yao, K. Ssu, and W.K. Fuchs, Message Logging in Mobile Computing, Proceedings of the 29th International Symposium on Fault Tolerant Computing, pp. 294-301, June 1999