Fault Tolerance within Session Initiation Protocol

Ekta Khurana

A Thesis
in
The Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science in
Electrical and Computer Engineering at
Concordia University
Montreal, Quebec, Canada

August 2005

Your file Votre référence
ISBN: 0-494-10238-1
Our file Notre référence
ISBN: 0-494-10238-1

CONCORDIA UNIVERISTY

School of Graduate Studies

This is to certify that the thesis prepared

By: Ekta Khurana

Entitled: Fault Tolerance within Session Initiation Protocol

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair

______________________________ Examiner

______________________________ Examiner

______________________________ Supervisor

Approved by ______________________________
Chair of Department or Graduate Program Director

____________20___ ______________________________
Dean of Faculty

ABSTRACT

Fault Tolerance within Session Initiation Protocol

Ekta Khurana

The Session Initiation Protocol is an application layer signaling protocol designed for the session establishment between end users. These SIP sessions are established based on a *request-response* paradigm between the client and server components of the user agent. Proxy servers may exist within the network, which serve as a bridge between two user agents who are either within different domains or are unregistered and unaware of each others' location. The entities however are not always reliable and may encounter faults. First, this thesis aims to identify the locations within a SIP session where faults can occur and the effect on the system. The current mechanisms used to handle these faults were analyzed and it was determined that the user agents were forced to recover from faults by means of restarting the session.

A novel approach was then proposed which detects and recovers from faults dynamically. This is achieved through the addition of a Decider block within the proxy domain which maintains the status of the proxy and determines if the message is to be sent to the primary or to its corresponding backup server. The fault detection is done through the usage of *Alive* messages sent from the proxy servers to the Decider block. This removes the recovery task from the user agent and reduces the dependence on a timeout mechanism. The proposed model is checked for correctness using the Specification and Description Language (SDL). Finally, after analysis of the message sequence charts generated from the simulation tool, the proposed model is verified.

Acknowledgements

I would like to take this opportunity to thank my supervisor Dr. Anjali Agarwal for her continuous guidance in carrying out this research. She has given me invaluable insight in my research domain while challenging me to step further as a means of attaining my research goals. I would also like to thank Mr. Nikhil Varma who has given important suggestions throughout the completion of this thesis. I would further like to express my appreciation to the examiners Dr. O. Ormandjieva and Dr. A. En-Nouaary for their comments for improving the presentation of this thesis.

Words are not sufficient to express the love and support that I have received from my parents Kul and Ritu and my sister Puja. They have given me constant courage and strength and they are the primary reason that I am able to accomplish this endeavor.

All of my friends in particular Uma Bharathi Ramachandran who have always given me a shoulder to lean on and have forced me to have an immense amount of enthusiasm and determination throughout my work. They have taught me the meaning of life and have supported me endlessly.

Table Of Contents

List of Figures

List of Tables

Chapter 1 Introduction

In this chapter, we begin with an overview of the Telephony and Signaling protocols present today. We then introduce the reason for which we have chosen SIP (Session Initiation Protocol) as our main focus. This leads into the discussion of the reliability of SIP and the need for fault tolerance within the system. We analyze the fault tolerance algorithms which can be applied to the protocol as a means of increasing its reliability. Finally, we state our motivation for the thesis and the contributions which we have made within this thesis. Throughout this thesis, the terms fault(s) and failure(s) are used interchangeably to represent the deviation of the system from the expected specification.

1.1 Signaling Protocols

Previously, telephony services were provided over circuit switched networks which used traditional phones. Within the last decade however, there has been an increase in sending audio, video and voice media over the Internet which is known as *Voice over IP* (VoIP) [1]. This has driven the need for reliable, scalable and interoperable signaling protocols which will allow this information to be transmitted successfully. The result of this need has been the specification of various protocols such as H.323 and SIP [2] which initialize connections between end users residing in local or wide area networks.

H.323 [3] is a recommendation developed in 1996 by the International Telecommunication Union (ITU-T). It combines several network elements and protocols as a means of allowing multimedia communications system over IP networks. The entities found in an H.323 network include:

- H.323 Terminal which is an endpoint within the network and provides two-way real-time communication with other terminals.
- H.323 Gatekeeper is an optional entity however serves two purposes. The first is that it provides admission control for the network determining which communication should be allowed between entities. The second is that it provides an address translation service between protocols.
- H.323 Gateway provides conversion between H.323 entities and non H.323 networks.
- H.323 Multipoint Control Unit which allows conferencing between three or more endpoints within the network. It has two subcomponents, the Multipoint controller which handles call control and the Multipoint processor which handles media exchange between the endpoints.

These entities use several protocols in order to actually set up the communication path and transport the media. The transportation of messages is conducted using the known Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

A simpler, text based protocol created as an alternative to H.323 is the Session Initiation Protocol (SIP) [4]. This application layer protocol designed by the Internet Engineering Task Force (IETF) was created in order to establish and modify sessions across networks. The main entities found within a SIP network are the following:

- SIP User Agent (UA) which is similar to an H.323 Terminal since it is an endpoint within the network and provides two-way real-time communication with other user agents. These contain both a client (UAC) and a server (UAS) component enabling it to send and receive messages.

- SIP Server which is similar to an H.323 Gatekeeper since it is an optional entity within the network. It provides information to UA with respect to the entities within the network and it also forwards messages on their behalf.

Similar to H.323, the transportation of messages is executed over TCP or UDP.

1.2 Why Choose SIP?

Although there are several solutions to sending media over IP networks such as the Internet, SIP has recently received a lot of importance since it encompasses both the call signaling and control mechanisms in a single protocol. This protocol is at the application layer and will therefore lead to a more reliable and more efficient support for real time communication. SIP also has several advantages over other protocols. Some of these advantages are the following [5, 6, 7]:

1. The setup time in SIP is less time consuming than other VoIP alternatives. This is due to the fact that the Session Description Protocol (SDP) [8] is used to determine the media capabilities of endpoints that are attempting to connect. The capability check is thus incorporated in the initial invitation message. If the features of the session are not feasible by the end user receiving the invitation, then the session will terminate prematurely thereby saving the resources required to establish and then terminate a session.
2. The encoding of messages within SIP is done using a simple text based mechanism. This increases the comprehensibility of the messages sent to the receiving end user

agents. Moreover, intermediary servers, if any, can easily determine the location of end users since the format is similar to that used for email addresses.

3. SIP is easier to implement since the functionalities are placed in a single protocol. Media exchange is achieved using the Real-time Transport Protocol (RTP) [9] however the call signaling capabilities required for session establishment are built into SIP.
4. SIP is both flexible and customizable since features can be added to a session using the header fields of the messages. Using the header fields allows the other entities within the session to know which type of messages will be sent. If they cannot process those Requests or Responses, they can inform the sending end user immediately. This assures consistency between the SIP entities with respect to the message types which are to be exchanged within the session.

1.3 Overview of Fault Tolerance

Through our discussion, it is evident why SIP has become so popular in the present VoIP market. However, signaling protocols should also be able to sustain even in the presence of faults within the system. Fault Tolerance is defined as the ability to respond positively to hardware or software faults within a system. There are many levels defined within Fault Tolerance [10] which enable us to determine the recovery procedures required when the fault occurs. They are described as the following:

- *Level 0 - No tolerance to faults*: In this level, when a fault occurs, the system dies and must be manually restarted from an initial safe state. Resulting data may be incorrect or inconsistent due to a time lag incurred due to the restart process.

- *Level 1 - Automatic detection and restart:* In this level, when a fault occurs, the error is detected and the system is restarted on the same server or on an alternative backup server, if any. As in Level 0, the resulting data may be incorrect or inconsistent due to the incurred time lag resulting from the restart process.
- *Level 2 - Level 1 including periodic logging and recovery of the initial state:* In this level, periodic states in the system are logged so that when a fault occurs, the error is detected and the system is restarted on the same server or on an alternative backup server, if any but from the last saved state. This eliminates the overhead of starting from the initial state.
- *Level 3 – Continuous operation without any interruption:* In this level, the system functions with the highest degree of availability. There is no fault which causes the system to fail and there is no situation in which the system will have to restart.

The two most used fault tolerance algorithms are the State-Machine approach and the Primary-Backup approach.

The State-Machine approach [11] "is a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with server replicas." The main idea within this approach is to have replications of the state machine being executed at the same time but on different processors. Each of the state machines receives the messages and processes them in the exact same order. This ensures that all of the state machines change states at the same time assuming that the processor controlling these state machines is non-faulty. Thus implementing fault tolerant state machines involves Replica Coordination [12] in which all replicas receive and process the same sequence of requests. This is divided into two sub parts namely:

- Agreement: Every non-faulty state machine replica receives every request. This is achieved if the protocol satisfies that
    a. All non-faulty processors agree on the same value
    b. If the transmitter is non-faulty, then all non-faulty processors use its value as the one on which they agree.
- Order: Every non-faulty state machine replica processes the requests it receives in the same relative order. This can be satisfied by assigning a unique identifier to the requests and having the state machine process the next stable request with the smallest unique identifier.

In this scheme, a voting mechanism is used to decide the common value between state machines. Therefore, the effects of the failures are completely masked. Although this fault tolerant mechanism is useful for many systems, it poses an overhead since each replica must receive the requests in order for the state machines to execute at the same pace. Thus a lot of resources are consumed within the system.

The alternative solution is to designate one server as the Primary and all others as the Backups. This is known as the Primary-Backup approach [12]. This technique assigns a particular server as a primary and all redundant servers as backups. The backup will receive periodic notification messages from the primary server in order to update the backup of the current state of the session. This continues until the primary server fails.

At this point, the pre-selected backup will take over all of the session functionalities to avoid any harm to the sender or receiver during the session.

The four properties of the primary backup fault tolerant approach [12] involve assuring that at most one primary or one backup server is servicing the requests. These properties include:

a. There exists a local predicate $Pmry_s$ on the state of each server s. At any time, there is at most one server s whose state satisfies $Pmry_s$.
    i. This implies that single primary server exists for each SIP client.
b. Each client i maintains a server identity $Dest_i$ such that to make a request, client i sends a message to $Dest_i$.
    i. This implies that the client sends the request to a single destination. In the case of a SIP client, the request is sent to a single proxy server.
c. If a client request arrives at a server that is not the current primary, the request is not queued (and therefore is not processed).
    i. This is to ensure that upon recovery, the primary and the backup do not respond to the same request.
d. There exist fixed values k and Δ such that the service behaves like a single (k, Δ)
    i. This implies that this solution can be used to implement those services which tolerate a bounded number of faults over their lifetime.

1.4 Need for Fault Tolerance in SIP

Subsequently after identifying the types of fault-tolerance mechanisms available it becomes clear that there is a need for implementing fault-tolerance in SIP. Currently, the

SIP sessions are not capable of handling all types of faults which occur within the system. There are several types of faults which can occur and these include link, user agent and server faults.

The SIP protocol contains three layers [4] through which the session is established. The lowest layer is the syntax and encoding layer which defines the rules for the header values. The middle layer is the transport layer which handles the link failures and manages the connections between end users. The third layer is the transaction layer which handles the creation and retransmission of requests and responses.

The connection that is shared between end users must be monitored for the destination IP address, the port and the transport protocol, either TCP or UDP, used within the session. These connections are generally kept open for the duration of the session establishment phase however if a link error occurs, the end users should be able to handle messages received from a different connection. The creation of a new connection is handled by the transport layer and is hidden to the SIP entities both end user agents and servers.

If any faults occur within the end user agents themselves, then the messages will cease to flow end-to-end. After a predefined time period, the end user which is still active will prematurely terminate the session. In turn, all resources allocated to this session will be relieved and a new session between two active end user agents can be established.

The most important fault type to consider is a fault in the server. Currently, there is no single mechanism implemented within SIP which allows the correct flow of a session with the presence of faults. There are several types of servers within SIP including the redirect, registrar and stateful and stateless proxy servers. In the case of the redirect, registrar and stateless servers, if a fault occurs, there is no harm to the session. All the messages sent to these servers have a prescribed time value attached to them and once the instance expires, the message is re-issued to an alternative server.

The main problem arises in the case of stateful proxy servers wherein the server wants to be aware of the SIP session at all times. At any point within the session, if the server fails, there must be a backup server available to take over or else the session will timeout and will have to be re-established.

Thus our goal within this thesis is to include fault-tolerance measures within SIP as a means of conserving the session resources previously allocated. Furthermore, we want to reduce the system's reliance on the timeout mechanisms built into SIP in order to reduce the elapsed time of the overall session.

1.5 Contribution

Research is currently in progress towards the inclusion of fault tolerance within SIP. The objective of our work is to determine the limitations in the existing fault tolerance approaches to SIP and to propose an enhanced fault tolerant scheme. The main contributions of this thesis are as follows:

(1) A new framework for fault tolerance within SIP has been developed. A redundant server has been added to take over the session in the presence of faults. Furthermore, a decision maker has been added which determines which server is active within the session. This removes the burden from the user agent to the actual proxy domain.

(2) Simulation of the proposed approach with SDL, a verification and validation tool for communicating systems. The tool generates Message Sequence Charts (MSC) which exemplifies the proposed algorithm.

(3) Fault-Tolerance within SIP has been demonstrated using the proposed fault tolerance algorithm under different failure scenarios within a session. This establishes that the proposed framework has potential for handling the occurrence of faults within the system.

1.6 Organization

The thesis is structured as follows: Chapter 2 provides an overview of the SIP protocol including the types of entities present within a SIP session as well as the messages exchanged. Chapter 3 discusses the current approaches to Fault tolerance in SIP and is followed by an introduction to our proposed approach. Chapter 4 introduces the SDL tool used to simulate our proposed approach. Further we describe the simulations conducted to assure that our fault-tolerant mechanisms function as expected. Chapter 5 describes our final simulation results and the verification of our SDL model against our proposed approach. Finally, in Chapter 6, we conclude with an overview of the work completed within this thesis and the possible future research work.

Chapter 2 Protocol Overview

In this chapter, we provide an overview of the fundamentals of the SIP protocol. We analyze the entities which communicate within a SIP session and the requests and responses issued for session instantiation and termination. We then discuss the extensions which have been added to the protocol in order to enhance its completion.

2.1 SIP Components

There are several components found within a typical SIP network. As described in [4] the end users are referred to as SIP User Agents (UA). They contain both a client component which sends requests and a server component which issues responses based on the capabilities of the network processing the requests. Furthermore, the UA have an embedded transaction user which creates the requests and responses to be sent through the client and server components. This user decides the message to be sent based on the flow of messages previously exchanged within a session. The original invitation message to a receiving end user is also generated from this transaction user. Therefore, the UA can instantiate a SIP session and can be both on the sending or receiving end.

In between UA, there exist several server entities required for different stages of a SIP Session. These servers are Registrar Servers, Redirect Servers and Proxy Servers. Registrar Servers are similar to a database in which they maintain the location of end users within their domain. At any point, if a sending user agent wishes to contact a receiving user agent, it can obtain the sending port from the Registrar server. These servers may also be integrated into the other servers within the network. The second type is a Redirect Server which supplies a new URI to the sending user agent without any further involvement in the session. This URI pertains to the location of the user agent for whom an invitation is intended to be sent. Finally there is a Proxy Server which serves as a bridge between two UA who are either within different domains or are unregistered and unaware of each others' whereabouts. These servers are used in order to process and forward requests from one end user to another end user. These requests may be recorded and responses issued from the Proxy which is the role of a stateful Proxy. On the other hand, the stateless Proxy simply forwards the request to the destination user agent. They are unaware of the status of the end users and that of the SIP Session in progress.

2.1.1 SIP Messages

The main SIP messages consist of requests or responses. These can be sent by any end user agent however the header field rules as described in [4] must be adhered to. Those messages which are incomplete will either be discarded by the components within the SIP network or responded to with an indication that the message could not be accepted.

2.1.1.1 Requests

There are several types of requests within the SIP Protocol [4]. These messages are only initiated by end UA however they can be processed by any Server component of the SIP Network.

The first request is INVITE which is used to instantiate a SIP Session. It contains an offer of the type of media the end user is expecting to transfer, the location from which it will be sending the media as well as the intended destination or called party. Once an INVITE request is sent, the user agent becomes the calling party for the duration of the SIP Session. This request is unique in that it is the only request that is outside of a SIP dialog and can be sent without any limitations.

The next request is CANCEL which allows any of the end users to prematurely terminate the session. This message can only be sent to terminate an INVITE request and is limited to being sent after receipt of a Provisional response and before receipt of a final Acceptance response from the called party.

The third is the ACK request which is sent by a calling party to notify the called party that the final response has been received. This message confirms the proper establishment of a SIP session between users and allows media transfer to commence. This request can also contain an answer to the offer supplied by the called party if not already done so in previous message exchanges between end users.

Termination of the SIP Session is achieved through the use of the BYE request. This both terminates the session and releases all resources allocated to the running transfer of media. This request can be generated by the calling party at any point within the session but can be sent by the called party only after an ACK request has been received. This limitation is to provide consistency with both end users.

The remaining requests are optional within the SIP Session. The first is the REGISTER request which allows an end user to register its location with the SIP Server. This information will be stored in a database and may be used by another end user within the same domain who wishes to establish a session and is unaware of the user's location.

The final request is the OPTIONS request which is used as a query to the server or another user agent to determine its capabilities. Since the OPTIONS request is simply an inquiry and has no effect on the state of the SIP Session, it can be sent both within and outside of a dialog.

2.1.1.1.1 Request Message Format

All requests within a SIP Session must begin with a Request-Line which states the name of the request (method name), the Request-URI and the SIP protocol version used. The message is then followed by header fields which will identify the requests within the session. These header fields are described in Table 2.1 [4].

An example INVITE request [4] containing both mandatory and optional header fields is shown. The other requests are similar however their request-Line will be unique.

```
INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4bK776asdhds
Max-Forwards: 70
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
Supported: 100 rel
Require: 100 rel
```

Header Field	Requirement*	Description
To	M	States the logical location of the called party with whom a session is to be established.
From	M	States the logical location of the initiator of the request. This will be known as the logical location of the calling party in the case of an INVITE request.
CSeq	M	Command Sequence Number which is generated by the party creating the request and uniquely identifies the session request.
Call-ID	M	Identification Number for a given dialog. All requests within the dialog will be identified with this common value.
Max-Forwards	M	This value is used to control the number of hops a request can undergo. The value is an integer (recommended to begin at 70) and is decremented by each subsequent hop thereby identifying the number of hops available for the message.
Via	M	This field states the route taken by the message and the expected forward path. It contains the transport protocol (TCP or UDP) used, the client's host name and the port number where it wants to receive responses.
Supported	O	This field lists the SIP Extensions that the user agent is capable of supporting including PRACK and UPDATE. If the field is empty then no extensions are supported by this user agent.
Require	O	This header field informs the server of the type of extensions that must be understood in order for the request to be processed.

* M = Mandatory, O = Optional

Table 2.1 Request Header Field

2.1.1.2 Responses

There are three main types of responses within a SIP Session namely Provisional responses whereby the status of the current transactions can be monitored, Error responses which inform of the incapability of handling a given session or Final responses which terminate transactions by means of an acceptance or rejection of the request received. The responses are classified into six main categories which will be elaborated upon in section 2.1.1.2.1 however there are several main responses that are transferred within the establishment of a SIP Session.

The first is the 100 TRYING Provisional response which is sent by a stateful Proxy Server only as a response to the INVITE request. This response is sent hop-by-hop and notifies the end user or the Proxy of the previous hop that an attempt is being made to contact the called party. Therefore, it assures that the INVITE Message is not retransmitted. Note that this is the only response that is not forwarded when received by a Proxy Server.

The next response is the 180 RINGING Provisional response which is instantiated by the called party as a means of alerting the calling party that a SIP Session may be possible. This response is sent end-to-end and may be retransmitted reliably depending on the use of the SIP extensions described later.

The other response is the Error response which informs the calling party that the session is not possible. The cause of this error can be due to the incapability of the called party, erroneous request header parameters sent to the servers or else general global errors within the network.

The 200 OK to INVITE Final response is the last main response found in SIP Session establishments. This response is often referred to as the final acceptance response since this assures that the called party will be partaking in the SIP Session. This response also contains an answer to the offer provided by the calling user agent in the INVITE request. If no offer was provided, the called party will extend the offer in this response and await an answer in the ACK request.

2.1.1.2.1 Response Classification

The responses within a SIP Session are categorized within six classes. Each of the classes contains several messages pertaining to the possible situations which may arise within the SIP Session establishment phase. Table 2.2 [4] describes the classes including a set of examples for each class.

Class Type	Class Description	Examples
1xx – Provisional	request received, continuing to process the request	180 Ringing, 181 Call Is Being Forwarded
2xx – Success	The action was successfully received, understood, and accepted	200 OK
3xx – Redirection	Further action needs to be taken in order to complete the request	300 Multiple Choices, 302 Moved Temporarily
4xx – Client Error	The request contains bad syntax or cannot be fulfilled at this server	400 Bad Request, 483 Too Many Hops
5xx – Server Error	The server failed to fulfill an apparently valid request	502 Bad Gateway, 505 Version Not Supported
6xx – Global Failure	The request cannot be fulfilled at any server	600 Busy Everywhere, 606 Not Acceptable

Table 2.2 SIP Response Classification

2.1.1.2.2 Response Message Format

All responses within a SIP Session must begin with a Status-Line which states the SIP protocol version used, the Status Code indicating the class of response and the Reason Phrase containing a description of the Status Code such as the examples shown in Table 2.2. The remaining header fields are the same as the mandatory fields in the SIP requests.

An example 200 OK response [4] is shown below. The other responses are similar however their Status-Line will be unique.

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP server10.biloxi.com
Via: SIP/2.0/UDP bigbox3.site3.atlanta.com
Via: SIP/2.0/UDP pc33.atlanta.com
To: Bob <sip:bob@biloxi.com>;tag=a6c85cf
From: Alice <sip:alice@atlanta.com>;tag=1928301774
Call-ID: a84b4c76e66710@pc33.atlanta.com
CSeq: 314159 INVITE
```

2.1.1.3 Basic SIP call flow

Several phases exist within a SIP Session. These include the Establishment phase, Termination of the session, UA Registration, Cancellation of the session and Query for the capabilities of a UA. The basic call flow of all of these phases within a SIP session is shown in Figure 2.1. The response to the INVITE request can be either a Provisional response or an Error response. Each SIP session does not need to include all of these phases however the initial INVITE message must be sent before any of the other phases occur. The other responses will have the same effect to the SIP session and are therefore not repeated.

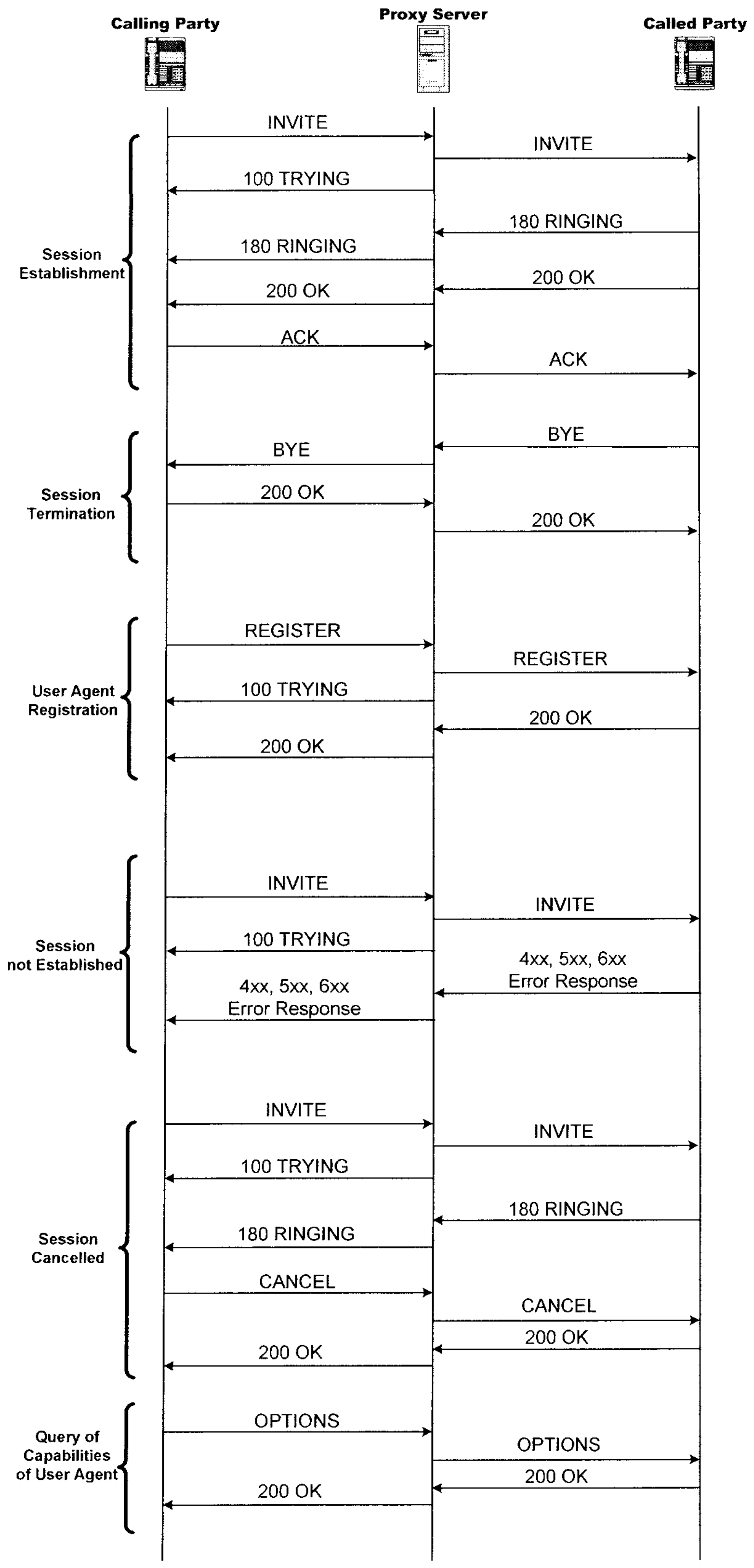


Figure 2.1 Basic SIP Message Transfer

2.1.2 SIP Extensions

The current set of SIP messages allows for the creation, modification and termination of a SIP Session. This allows media communication between end users. However there exist several scenarios which cannot be handled with this set of requests and responses. Therefore, several extensions have been established as a means of providing a solution to a specific requirement within the SIP protocol. These are described in the following sections.

2.1.2.1 PRACK

Within the SIP Session, the 200 OK response is delivered reliably since it is retransmitted until an ACK request is received. This assures that the calling party is aware of the interests of the called party. Similarly, the calling party uses the Provisional response such as 180 RINGING to be aware of the status of the called party. Without this Provisional response, end users may misinterpret the called party and terminate the invitation to a SIP Session prematurely. However, there is no enforced reliability for the Provisional responses. Thus, to eliminate this problem, the PRACK extension [13] was introduced to the SIP Protocol.

Once the Provisional response is received by the calling party, it issues a PRACK (Provisional response Acknowledgement) in order to notify the called party of the receipt. This causes the called party to stop retransmission of the Provisional response. Finally the called party issues a 200 OK to PRACK message which completes this extensions' message exchange.

The reliability of the Provisional response may not always be necessary and therefore, the use of this extension is decided by the calling party. If the reliability is to be enforced, the calling party should set the proper header fields in the initial INVITE request. The optional header fields Supported and Require should be set to 100rel if this extension is to be used. This indicates to the called party that reliable Provisional responses must be sent. If this request is not supported by the calling party, a 420 Bad Extension Error response should be issued to the INVITE request. If the Require header field is null, then Provisional responses will not be reliable and PRACK will not be issued. This extension thus ensures reliable Provisional responses however the usage is optional.

2.1.2.2 UPDATE

The calling party initiates a session with an INVITE request stating the offer of the session which includes parameters such as the type of media to be sent and the duration of the session. Amid a call however, either user agent may want to modify these boundaries. At present, the user agent must wait for a response to the original INVITE and then issue a re-INVITE containing the new desired session parameters. This consumes a large amount of time and moreover a lot of resources are wasted since the messages exchanged till this point will not be used for the remaining part of the session.

The solution is the introduction of the UPDATE [14] extension which allows the UA to change the session parameters midway through without reissuing an INVITE request. This request can be sent by either end user and contains a new offer. In response, a 200 OK to UPDATE response is sent which contains the answer to the changed parameters (i.e. whether the other party is capable of handling this change). The UPDATE Extension can only be utilized before the transfer of the 200 OK to INVITE Final Acceptance response. If this response has already been issued, then a re-INVITE must be sent. Furthermore, for the usage of this extension, the initial INVITE must contain an Allow header field stating UPDATE. This gives permission to the called party to change the session parameters midway.

The message flow of PRACK and UPDATE are shown in Figure 2.2. Note that it is assumed within this message flow that the header fields are set to the required values.

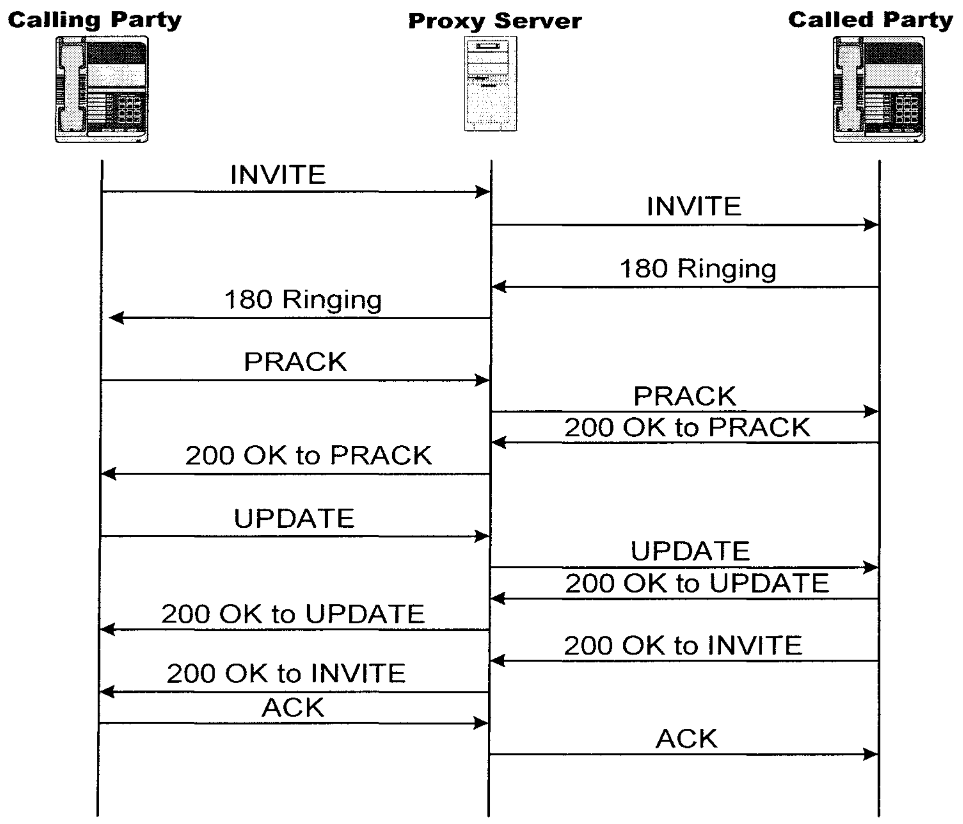


Figure 2.2 PRACK and UPDATE Message Flow

2.1.2.3 SUBSCRIBE AND NOTIFY

The previous extensions alter the instantiated SIP sessions however the SUBSCRIBE and NOTIFY extension [15] is used to notify another SIP entity of a running session. Therefore, this extension has no effect on the running dialog between end users. The subscription can be used to find out about the present status and/or any change in the state of a resource. The subscriber sends a SUBSCRIBE message to the entity which it wishes to track. SUBSCRIBE requests contain an "Expires" header which indicates the duration of the subscription. The Subscribe message must also have an “Event” header amongst the header fields which will specify the type of event it is subscribing to.

Upon acceptance of this message, the entity becomes the notifier. The notifier sends NOTIFY messages to inform subscribers of changes in state to which the subscriber has a subscription. This extension can be used by both UA and servers and assures that the subscriber is aware of the current state of the notifier. To unsubscribe, a SUBSCRIBE message with the “Expires” field set to zero is sent to the notifier. After this point, NOTIFY messages will no longer be sent.

The basic flow of messages for the SUBSCRIBE and NOTIFY extension are as illustrated in Figure 2.3 [15].

2.2 Summary

This chapter began with an overview of the Session Initiation Protocol. We have introduced the components found in a SIP network including the end UA, the Registrar,

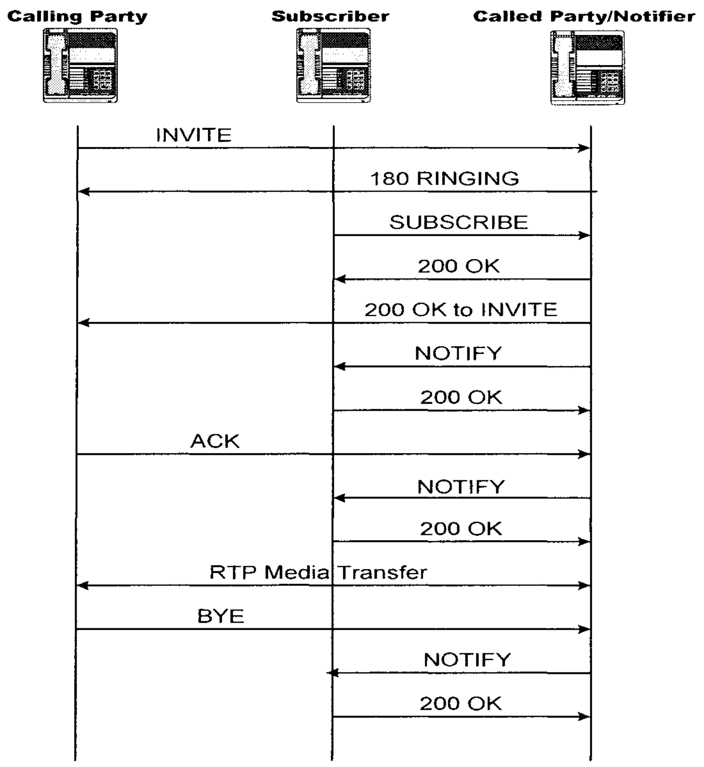


Figure 2.3 SUBSCRIBE and NOTIFY Message Flow

Redirect and Proxy servers. We then discussed the messages which can be sent between UA namely in the form of requests and responses. We have introduced the Header Formats required for these messages to be processed by the SIP elements and the classification of these messages within the SIP message exchange set.

This discussion further led to the SIP extensions and the limitations which they resolved. The extensions include PRACK which allows for reliable Provisional responses, UPDATE which allows the session parameters to be changed before the session is established and finally SUBSCRIBE and NOTIFY which allows another SIP element to be aware of the status of a running session without being a member of the actual message exchange.

Chapter 3 Fault Tolerance within SIP

In this chapter, we discuss which entities can be unreliable and the effect on the session establishment. We then discuss the current approaches used to achieve Fault Tolerance within a SIP network. We survey their methodology as well as the types of faults which are handled within their system. We then discuss our analysis of a complete SIP Session including both the establishment and termination phases. We identify the types of faults which can occur within the session and at the precise location at which these faults occur. Finally we describe in detail, our proposed approach for Fault Tolerance within SIP. This includes both the system level overview and a detailed message flow description.

3.1 Fault Tolerance Issues within SIP

The SIP protocol is advantageous for media transfer and allows mobility within a system of networks. On the other hand, this protocol is not designed to handle large amounts of traffic. For this reason, the resources which SIP utilizes must be able to cope with the failure situations which may arise without compromising the service it is expected to provide.

Failures within the system may be due to proxy server failures or failures in the links along which the requests and responses are sent and received. In both cases, the initial phase is fault detection whereby the location of the fault is known and secondly there is the fault recovery phase where the situation is handled to avoid system shutdown and to resume correct protocol behavior.

3.1.1 Link Failures

The SIP protocol contains three layers through which the connection is established. The middle layer is the transport layer which describes the details of how a client will send requests and receive responses and how a server will receive a request and send a response over the network. At this layer, the link failures which can occur are twofold. The first is a failure in the link between two user agents. The second is a failure in the link between the user agent and the server. Generally, the transport layer sends and receives messages along the same connection. In the case of link failures however, the transport layer will open a new connection which the end users and the servers should be capable of understanding.

3.1.2 Proxy Failures

In the case of a proxy server, two possibilities are present. If the proxy is a stateless proxy, its simplistic functionality within the session is to forward the requests from the sender to the intended receiver. They are simple and fast and can be used as simple load balancers or routers. One of the main drawbacks of the stateless proxy is they are unable to retransmit messages or perform any advances routing such as forking. If this server fails, no response will be received by the sender and a timeout on the request will occur. The sender can then detect that the server has failed and it will reissue the request to another stateless proxy server. This failure will not damage the functioning of the system however it will incur an additional delay since the request will have to be reissued.

For the stateful proxy case, there are additional concerns since the actual state of the session is stored and the proxy is involved in the actual responses rather than simply forwarding the information. This assures that the stateful proxy can handle retransmissions (if the message is already received) and they can issue several messages in response to the receipt of a single message. Although stateful proxy servers are more complex, they are widely used since they offer accounting and retransmission features. Due to the fact that the state is maintained, if retransmission mechanisms are used, the alternative servers must be notified of the current status of the session before they can successfully take over the session.

3.2 Current Approaches

Achieving Fault Tolerance within SIP is essential since it assures the proper delivery of messages and media exchanged between end users. As mentioned earlier, both link and server faults can occur. The Link faults are handled by the transport layer of the SIP layer sequence. Therefore, the primary faults to consider within the SIP session are Server faults.

The first stage is fault detection wherein a system is diagnosed and the occurrences of faults are identified. These faults can be transient faults wherein the server becomes temporarily unavailable and then returns to a safe state or else permanent faults where the server is no longer available for the entire SIP Session. In both cases, the session should perform as expected for both the Establishment and Termination Phases. The tolerance of these faults can be achieved by placing the load on the end users or the servers themselves. Several authors have analyzed this issue and through means of timeouts and replication they have achieved a means to detect and handle faults respectively.

3.2.1 DNS Query

Achieving Fault Tolerance within a SIP Session can be done through means of a replicated server. When a user agent sends a request, a timer is started and if no response is received within the prescribed time, the request is retransmitted to the replica server. The selection of this alternative server is determined through means of a Domain name space resource record (DNS-RR) lookup where the DNS server's primary role is to translate a domain name to an IP address [16,17,18]. The server resource record (SRV-RR) states the services offered by a given domain and also allows a single domain to have several servers contained within it. The location of these servers is known when performing a lookup of a particular domain. Moreover, these servers are assigned a priority value which will enforce the pattern in which these servers are accessed. The user agent who issues the request will have access to a database of these servers in the following format

_Service._Proto.Name TTL Class SRV Priority Weight Port Target

This approach uses the timeout mechanism built into SIP as a means of Fault Detection. Once the timeout occurs, the user agent will attempt to find another server which can send the request. An example of the information visible to the user agent for the domain example.edu is:

```
_sip._udp.example.edu  SRV 0 5 80 p1.example.edu
                       SRV 1 5 80 p1.example.edu
```

The advantage of this scheme is that the location of the alternative servers is known to the user agent however the logic need not be built into its system. The disadvantage however is that the burden of finding an alternative path is still placed on the user agent and moreover, the timeout mechanism will delay the entire session establishment.

3.2.2 RSerPool

An alternative to tolerating faults is to maintain a dynamic list of available servers which are contained within a reliable pool of servers [19, 20]. The concept behind this approach is to provide high reliability and increased scalability by means of choosing an available server dynamically without waiting for the timeouts to occur. Conrad et al. propose a method to dynamically determine the liveness of a server prior to the issuance of a request while still maintaining redundancy as achieved with the DNS query.

The working of this proposed solution is that each server is considered to be a pool or set of servers. Each pool of servers is located within a common domain and the servers can be accessed by the calling user agent when a request is issued. Periodically, the user agent can use the ENRP (Endpoint Name Resolution Protocol) server to detect the current status of a SIP Element. This prevents sending requests to a server that has permanently failed. Moreover, detecting the fault within a SIP server in advance eliminates the dependency on a timeout mechanism. This removes the additional delay placed on a SIP Session when using the previous DNS query approach. Although the process is dynamic, the drawback is still that the load of fault detection is placed on the SIP user agent.

3.2.3 State Sharing Algorithm

The previous approaches discuss the manner in which to obtain an alternative server when a faulty server within the network is detected. Thus, Fault Tolerance can be achieved through the usage of server replication within the SIP Session. However, simply replicating the server is not sufficient since the replicated copy must be aware of the current SIP Session even while it is dormant. For this, Bozinovski et al. [21] have proposed a state sharing algorithm which sends a text based update message as a means of notification to the replicated servers. This message is sent using the trivial File Transfer Protocol (TFTP) at a lower level than the running SIP Session. This provides the replicas with the current state of the server which has transferred requests or responses within the SIP Session. This allows all servers within the system to be consistent.

Prior to the usage of replicas however, the fault must be identified. Fault detection is executed by means of a timeout mechanism. Requests are sent to a selected server and if no response is received within a pre-defined time, the request times out and is re-issued to another server. The alternative server is chosen using a server Selection Policy (SSP). The selection policy can be either static as in the case of DNS Lookup or dynamic as in the case of RSerPool.

The faults considered within this approach can be either transient or permanent however the distinction has not been explicitly stated by the authors. A server which has failed may be reselected if it satisfies the criteria set according to the SSP. In the case of permanent faults, the requests will constantly timeout however in the case of transient faults the requests may pass successfully given that the server has recovered from the fault.

This proposed method updates the other servers within the list of available alternatives however the dependence on the timeout mechanism delays the overall SIP Session. Furthermore, there is no differentiation between permanent and transient faults within the server therefore the faulty server will constantly send requests which will go undelivered. This wastes the session resources without cause and introduces an additional delay.

3.3 Need for a New Approach

The current approaches to achieving fault tolerance within SIP session all use the notion of redundancy. Replicating a server allows a means to fail-over to a safe state without terminating a running session. Nonetheless, a set of rules need to be defined between the replicated servers as a means of determining which server will receive the user requests and responses given that all servers are currently active.

The DNS Query method defines a priority but the other approaches do not introduce this aspect. The set of rules which we have chosen between our replicated servers is based on the primary backup algorithm as described in Section 1.3. At any given time a single primary proxy server exists. Furthermore, when the primary proxy server recovers from a fault, it must request the state from the backup server in order to assure that only one server will receive the messages at a time. This eliminates the double propagation of the same message.

The second requirement is to create an approach which provides fault tolerance within a SIP session without dependence on the UA. A UA should send or receive messages with the assumption that they are reliably reached end-to-end. The third requirement is that the level of fault tolerance be increased so that the system is not restarted from the beginning. After fault recovery, the system restarts dynamically from the current state of the session. Finally, in order for this approach to be fully functional, the messages must not be dependant on a timeout mechanism. If a timeout does occur, the request will have to be retransmitted to the backup server. The comparison of the current approaches is shown in Table 3.1.

	Dependency on Timeout mechanisms	Redundancy Used	Primary Backup Scheme Employed	Status of server known dynamic -ally	Burden on User Agent	Fault Tolerance Level (Section 1.3)
DNS Query	Yes	Yes	No	No	Yes	1
RSerpool	No	Yes	No	Yes	Yes	1
State Sharing Algorithm	Yes	Yes	Yes	No	Yes	1
Proposed Approach	No	Yes	Yes	Yes	No	2

Table 3.1 Comparison of Current Approaches to the Proposed Approach

3.4 Proposed Approach: Overview

Knowing wherein the SIP session faults can occur is essential in order to determine the recovery mechanism. However, in all solutions discussed till now, the redundancy although available must be identified and accessed by the end user agents. The Calling or Called parties determine the location of alternative servers, if any, and then retransmit their requests and responses as a means of assuring it reaches the intended destination.

Our approach removes the load from the user agent and places it within the server domain. This assures that the messages will reach the destination and moreover this removes the need for retransmission from the user agent. At any point within the SIP Session, the user agent can send a request or response and our proposed Fault Tolerance Block will ensure that it reaches its intended destination.

The second addition is that our approach does not use the timeout mechanism as a means of identifying the presence of faults. Instead, an *Alive* message is introduced which is sent in order to inform its domain that the proxy server is available to process the end user messages. This removes the additional delay incurred when the user agent waits for a predefined time to elapse before retransmitting the message to the backup server. Performance issues related to the addition of the *Alive* message are discussed in Section 4.6.

If a primary server fails or if an error occurs in its transmission the message is retransmitted to the known backup server. However, an issue which has until now been overlooked is the fact that the primary server may come back to a safe state. At this point, the backup server should relinquish all duties and pass them back to the primary server that is now capable of handling the SIP Session. Our approach assures that the Domain Manager is aware of the status of its servers at all times and thus the primary server will always be chosen so long as it is capable of transmitting the messages. Fault recovery is therefore dynamic and does not delay the overall session thereby increasing the fault tolerance level of our system.

Maintaining correct status with the Domain Manager is essential however the same must be done between the primary and backup servers. For this, our proposed approach uses the SIP Extension NOTIFY message as a means of assuring that the backup server is aware of the status of the primary server with respect to the current SIP Session. This allows the backup to dynamically take over the session at any point. The communication between servers also allows the primary server to be set back to the current state within the session when it recovers from the previously occurred fault.

3.4.1 Decider server

The Domain Manager is referred to as the Decider server and is located within every given domain. It is aware of the location of the primary server and all backup servers. The Decider server also knows about the status of each of the servers (i.e. whether or not it is capable of receiving requests or responses). If the server is alive, the state of the session is only known to the primary and backup servers since maintaining this information in the Decider server is unnecessary.

There are three main assumptions which we have taken within this approach. The first assumption is that the Decider server is reliable and that the links from the Decider are also reliable. This ensures that the Decider server will, at all times, be aware of the status of the primary and backup servers within the domain. The second is that between the primary and backup proxy servers, only one of the proxy's fails at a time. This ensures that the Decider block is aware of the status of both servers within the domain and more importantly that a backup server is available to process the message. The third main assumption is that when the primary proxy server sends a message, the notification is atomic and is completed before the fault occurs. This ensures consistency within the network and allows the backup to take over the job. However, if the primary proxy server sends a request or response and then the fault occurs before the notification, this will lead to an inconsistent state. The backup proxy server will not be able to recognize the next message which it receives and thus the SIP session would not be established.

3.4.2 SIP Architecture

Through usage of the primary backup approach and the inclusion of the Decider server, the elements within any given SIP Network include the end user agents referred to as the Calling and Called Party, the primary proxy server(s), the backup proxy server(s) and the Decider server. The finite state machine of each element is referred to as a process. The Calling Party process is P1, Called Party process is P4, primary proxy server process is P2, the backup proxy server process is P3 and the Decider server is P5. Each process runs in the individual server and the servers communicate as shown in the block diagram in Figure 3.1 and their communication channels are bi-directional.

The set of requests and responses transmitted between end users are shown in Table 3.1. The 100 TRYING message is labeled as m2a since it is found within the

Provisional response classification however it is a hop-by-hop message and is thus separated from the class of m2 messages.

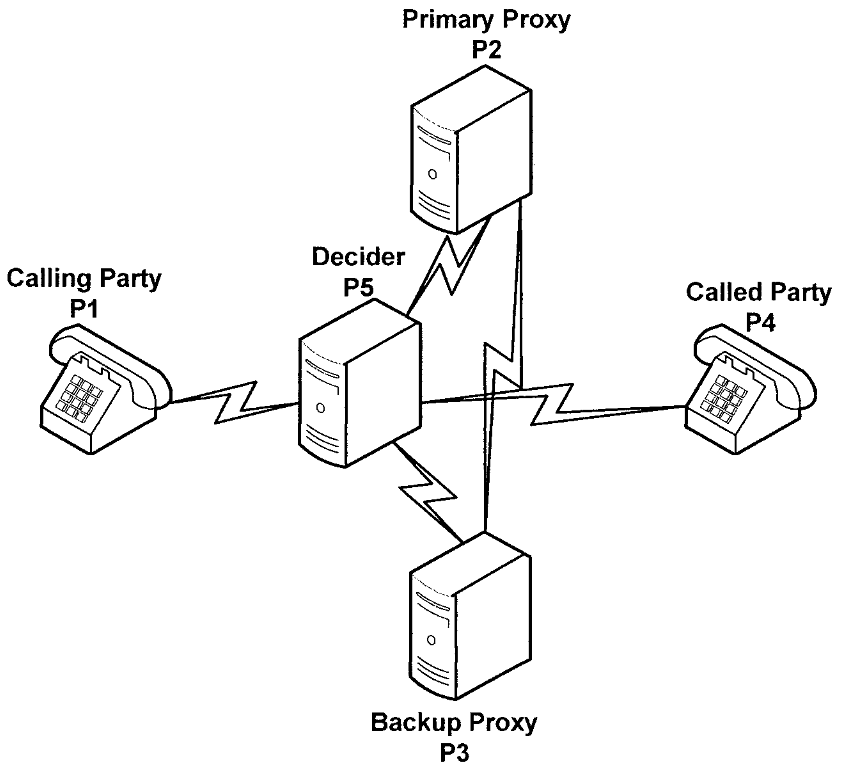


Figure 3.1 Process Communication within Proposed Approach

The highlighted messages are those introduced within our proposed approach. The first is the *Alive* message which is used by the proxy servers (both primary and backup) to notify the Decider server that they are functional and can issue messages between the UA. This message is sent to the Decider server at periodically. If no *Alive* message is received then within a defined time span a fault is detected and the backup is labeled as the current primary. All messages received will be issued to the backup.

Once the primary server recovers from a fault, it must determine the current status of the session. For this reason, the *StateRequest* message has been added which sends a request to the backup server to notify the state of the session. This assures that both the primary and backup are not active within the session at the same time. The final addition is the *StateResponse* message which is the response to the *StateRequest*. This message contains the current state of the session. Once the primary receives this message, it reads the contents and transfers to the given state. The primary then issues an *Alive* message to the Decider server and it can resume activity within the session.

The workings of the SIP elements inclusive of our proposed Decider server are shown by means of finite state machines. The convention used is that each state machine contains a queue to send messages (represented by -) and a queue to receive messages (represented by +).

m1	INVITE
m2	PROV response
m2a	100 TRYING
m3	Error response
m4	PRACK
m5	200 OK to PRACK
m6	200 OK to INVITE
m7	BYE
m8	200 OK to BYE
m9	ACK
m10	NOTIFY
m11	**Alive**
m12	200 OK to NOTIFY
m13	**StateRequest**
m14	**StateResponse**
m15	SUBSCRIBE
m16	200 OK to SUBSCRIBE

Table 3.2 SIP request and response message set

A SIP UA contains both the functionalities of the Calling and the Called parties however for clarity we have separated them. Figure 3.2 illustrates the Calling Party. The

Calling party starts in an *Idle* state. It then initiates a session by sending an INVITE request which transfers the state machine to the *Calling* state. At this point, a 100 TRYING message is sent by the proxy of the next hop. This does not change the state of the Calling party however receipt of this message must be handled and is done so with a self-loop. Once a Provisional response is received, the Calling party goes to the *Attempt* state which means that the UA is attempting to establish a session. At the *Attempt* state, retransmissions of the provisional response may arrive since the PRACK extension is used and the message is assured to reach the end user reliably. These retransmissions are done by the Decider block and receipt of them is denoted as another self-loop. The Calling party then issues PRACK which takes the machine to a *ProvReceived* state indicating that the Provisional response has been successfully received by the Calling party. Once the 200 OK to PRACK response is received the Calling Party is in the *AlmostReady* state. Receipt of the 200 OK to INVITE message takes the Calling party to a *Pre_Connection* state and finally once the ACK is sent, the machine enters the *SIPSession* state and the session establishment phase is complete.

When a BYE message is sent, the Calling Party enters the *Wait_For_Acceptance* phase and finally when the 200 OK to BYE response is received, the Calling party enters the *NOSIPSession* state and the session termination phase is complete. As per the rules of the BYE message, it can be sent after session establishment or before receipt of the 200 OK to INVITE acceptance response namely the *AlmostReady* or *SIPSession* states of the Calling Party state machine.

If an Error Response is received from the Called party, the Calling party transfers to the *NOSIPSession* state indicating that the session has not been established.

Figure 3.3 illustrates the Called Party. The Called party starts in an *Idle* state. Upon receipt of an INVITE request the state machine transfers to the *Called* state. At this point, a Provisional response is sent to the Calling party and an intermediate state *Attempt* is reached. The Called party then waits for the PRACK message in order to go to the *ProvReceived* state which notifies the Called party that the Provisional response issued was in fact received by the Calling party. Note that no retransmissions occur within this state machine therefore no self-loops are required.

Next, the 200 OK to PRACK response is sent and the Called Party is in the *AlmostReady* state. Sending of the 200 OK to INVITE message takes the Called party to a *Pre_Connection* state and finally receipt of the ACK forces the machine to enter the *SIPSession* state and the session establishment phase is complete.

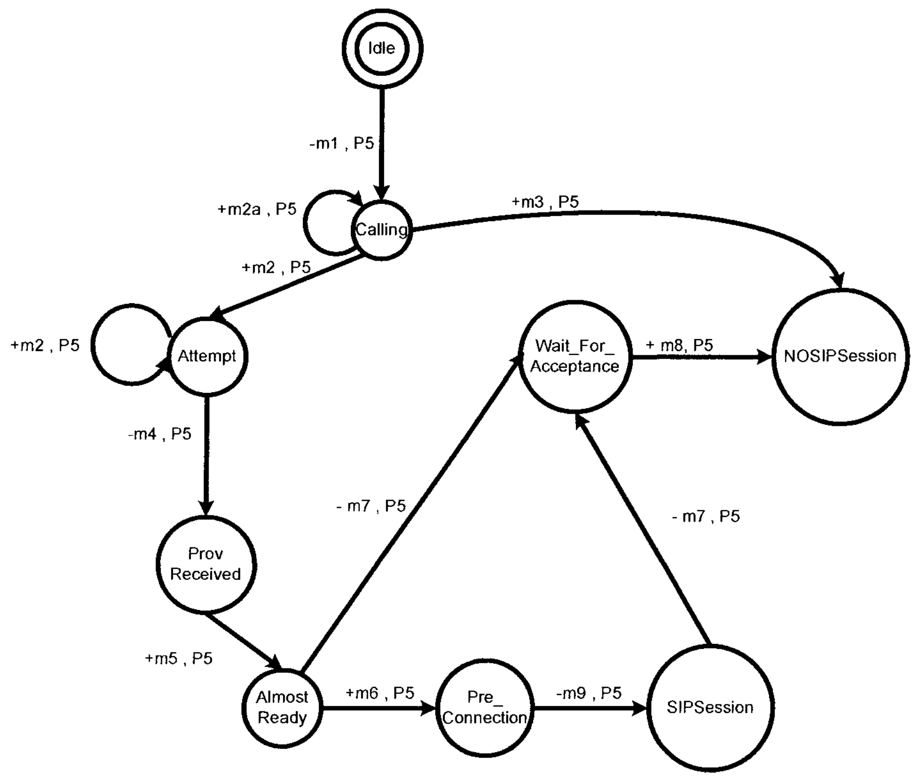


Figure 3.2 Calling Party (P1) State Machine Diagram

When a BYE message is received, the Called Party enters the *Wait_For_Acceptance* phase and finally when the 200 OK to BYE response is sent, the Called party enters the *NOSIPSession* state and the session termination phase is complete. The Called party must be able to receive the BYE message sent by the Calling party which corresponds to the *AlmostReady* and *SIPSession* states of the Called Party state machine.

If an Error Response is sent to the Calling party, the Called party transfers to the *NOSIPSession* state indicating that the session has not been established.

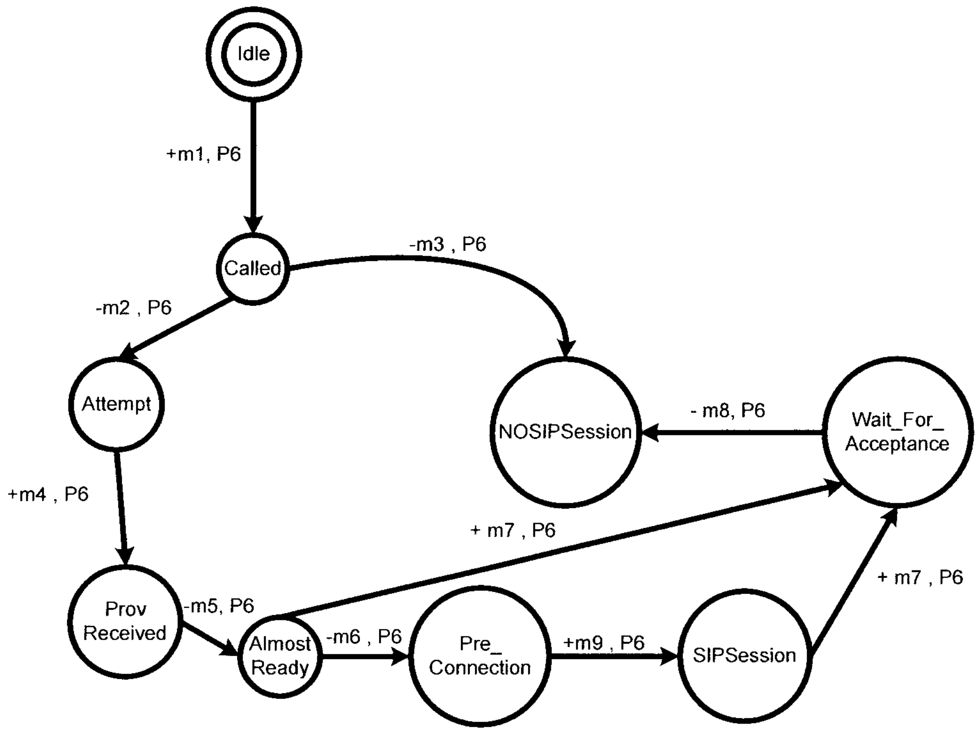


Figure 3.3 Called Party (P4) State Machine Diagram

Figures 3.4 and 3.5 illustrate the primary and backup proxy servers respectively. The primary server starts in an *Idle* state. Requests are received from the Decider server attached to the Calling party (P5) and sent to the Decider server attached to the Called party (P6). The responses traverse in the opposite direction from P6 to the proxy and then to P5. Within each state transition, the primary and backup communicate by means of a subscription and notification mechanism. At every state change (meaning every time a message is sent by the primary proxy), the primary issues a NOTIFY message. The backup then receives the notification, changes to the corresponding state and issues a 200 OK to NOTIFY indicating that it has successfully changed to the new state.

Note that the branch between states *Calling* and *Trying* will only occur if there exist two or more proxy between the UA. In that case, a 100 TRYING from the next hop will be issued to the primary proxy of the previous hop. The receipt of this message is shown along the transition labeled –m2a, P6. Furthermore, if the primary server fails it returns to the *Idle* state. Upon recovery, it can send m13 to the backup server to request the current state of the session. The state is sent in message m14. The primary server then reads the response and transfers to the given state.

Within the backup proxy server, the machine changes states based on the receipt of a request or response or by receipt of the notification message from the primary. Note that at every state, the state machine is able to receive a *StateRequest* message m13 and reply with the state of the session in m14. The primary server can recover at any point and thus the receipt of the state request must be possible throughout the backup proxy server state machine.

The Decider server connected to the Calling Party is shown in Figure 3.6 and the Decider server connected to the Called party is shown in Figure 3.7. Note that the message flow of the Decider connected to the Calling is the ***sender*** component while the

Decider connected to the Called party is the ***receiver*** component. These processes can be merged into a single server however they are separated for visual clarity.

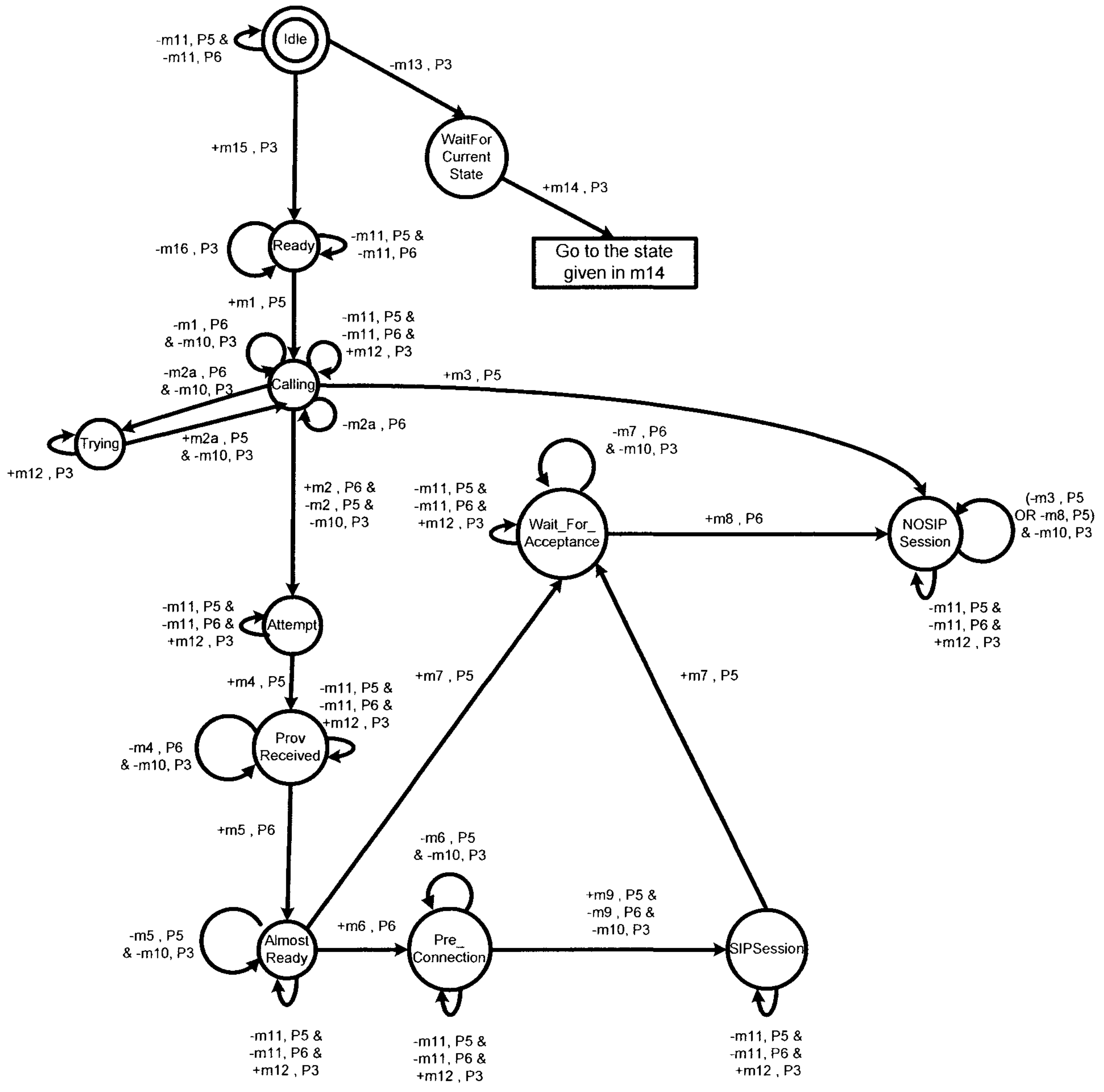


Figure 3.4 Primary proxy (P2) State Machine Diagram

The state machine of this entity also begins in an *Idle* state. If an *Alive* message is received from the primary server then the state machine goes to the *PrimaryIsAlive* state.

In this state, if the *Alive* message is received from the backup then it is ignored and no change in state occurs. This is denoted as a self-loop. At this state, if a request or response is received, it is sent to the primary server. If the backup *Alive* message is received first, the Decider goes to the *BackupIsAlive* state.

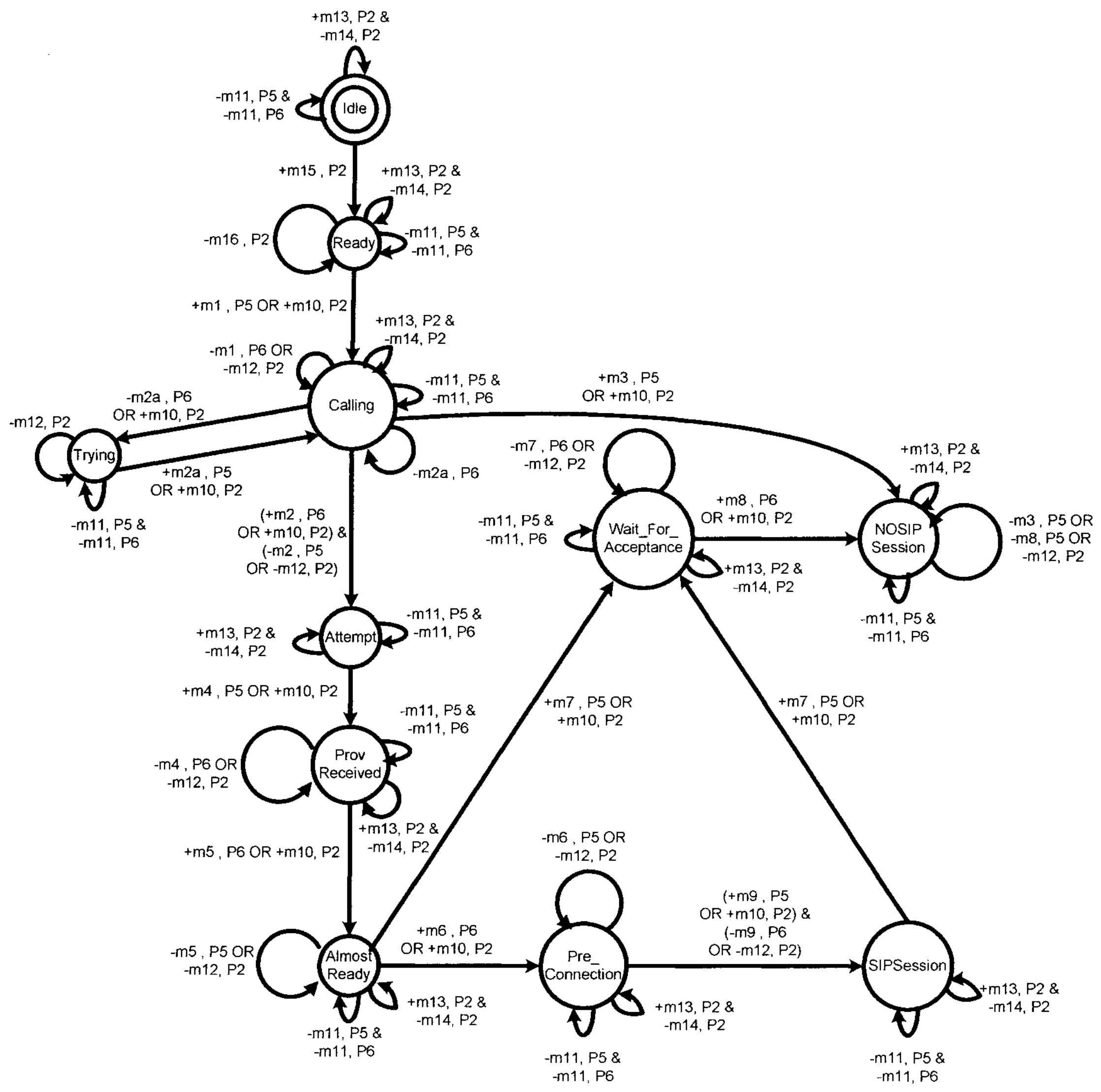


Figure 3.5 Backup proxy (P3) State Machine Diagram

If an Alive message is received from the primary then the machine reverts back to the *PrimaryIsAlive* state. This is required since we cannot guarantee the order in which the *Alive* messages will be received. While in the *BackupIsAlive* state, if no *Alive* is received from the primary within a predefined time lapse, then the request or response is sent to the backup server. This timeout may also occur when no message is received which is denoted by the self loop on the *BackupIsAlive* state. In this case, the Decider server will wait till it receives a message.

When the Decider server sends a message, it stores it into a buffer1 and returns to the *Idle* state. If the primary server fails after the message was sent to it, the message is lost and therefore, a retransmission to the backup server is required. A timer is placed on each message and if it expires (i.e. no response has been received), the Decider will detect that a fault has occurred on the primary proxy server. The Decider then reads the buffer1 which contains the previously sent message and resends it to the backup server. The size of the buffer1 is one. Moreover the buffer1 does not need to be flushed since it is overwritten each time a new request or response is sent.

The second buffer within the state machine, labeled buffer2 is required to save messages received by the Decider block before the retransmission of the previous message. This is needed to ensure that no messages are overwritten before their retransmission. This scenario occurs if the primary proxy server fails after the Decider has sent message m5 to it. At this point, the Called party may issue message m6 before m5 is retransmitted to the backup proxy server. To avoid this scenario, message m5 is stored in buffer1 for retransmission while m6 is stored in buffer2 and sent after buffer1 is empty. The size of the buffer2 is one. Moreover the buffer2 does not need to be flushed since it is overwritten each time a new request or response is received.

It should be noted that the finite state machines shown for the Decider blocks are for the scenario wherein the Deciders are connected to the Calling and Called agents directly. If additional proxy servers exist between the UA as shown in Figure 3.8 then P6 of the first proxy will send requests to P5 of the second proxy and the responses will traverse the opposite route from P5 of the second proxy to P6 of the first proxy.

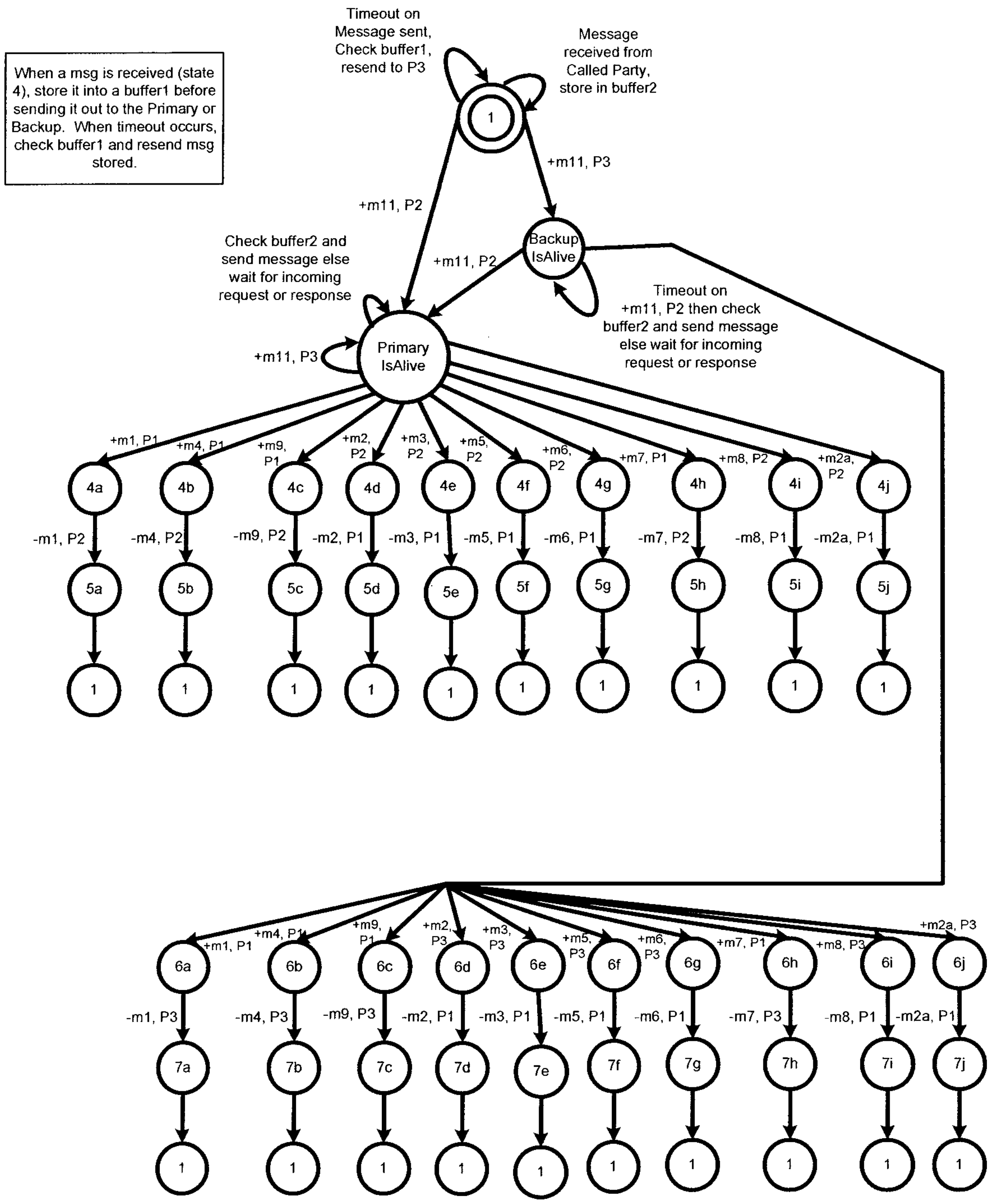

When a msg is received (state 4), store it into a buffer1 before sending it out to the Primary or Backup. When timeout occurs, check buffer1 and resend msg stored.
Timeout on Message sent, Check buffer1, resend to P3
Message received from Called Party, store in buffer2
1
+m11, P3
+m11, P2
Backup IsAlive
+m11, P2
Check buffer2 and send message else wait for incoming request or response
Timeout on +m11, P2 then check buffer2 and send message else wait for incoming request or response
Primary IsAlive
+m11, P3
+m1, P1
+m4, P1
+m9, P1
+m2, P2
+m3, P2
+m5, P2
+m6, P2
+m7, P1
+m8, P2
+m2a, P2
4a
4b
4c
4d
4e
4f
4g
4h
4i
4j
-m1, P2
-m4, P2
-m9, P2
-m2, P1
-m3, P1
-m5, P1
-m6, P1
-m7, P2
-m8, P1
-m2a, P1
5a
5b
5c
5d
5e
5f
5g
5h
5i
5j
+m1, P1
+m4, P1
+m9, P1
+m2, P3
+m3, P3
+m5, P3
+m6, P3
+m7, P1
+m8, P3
+m2a, P3
6a
6b
6c
6d
6e
6f
6g
6h
6i
6j
-m1, P3
-m4, P3
-m9, P3
-m2, P1
-m3, P1
-m5, P1
-m6, P1
-m7, P3
-m8, P1
-m2a, P1
7a
7b
7c
7d
7e
7f
7g
7h
7i
7j

Figure 3.6 Decider server connected to the Calling Party (P5) State Machine Diagram

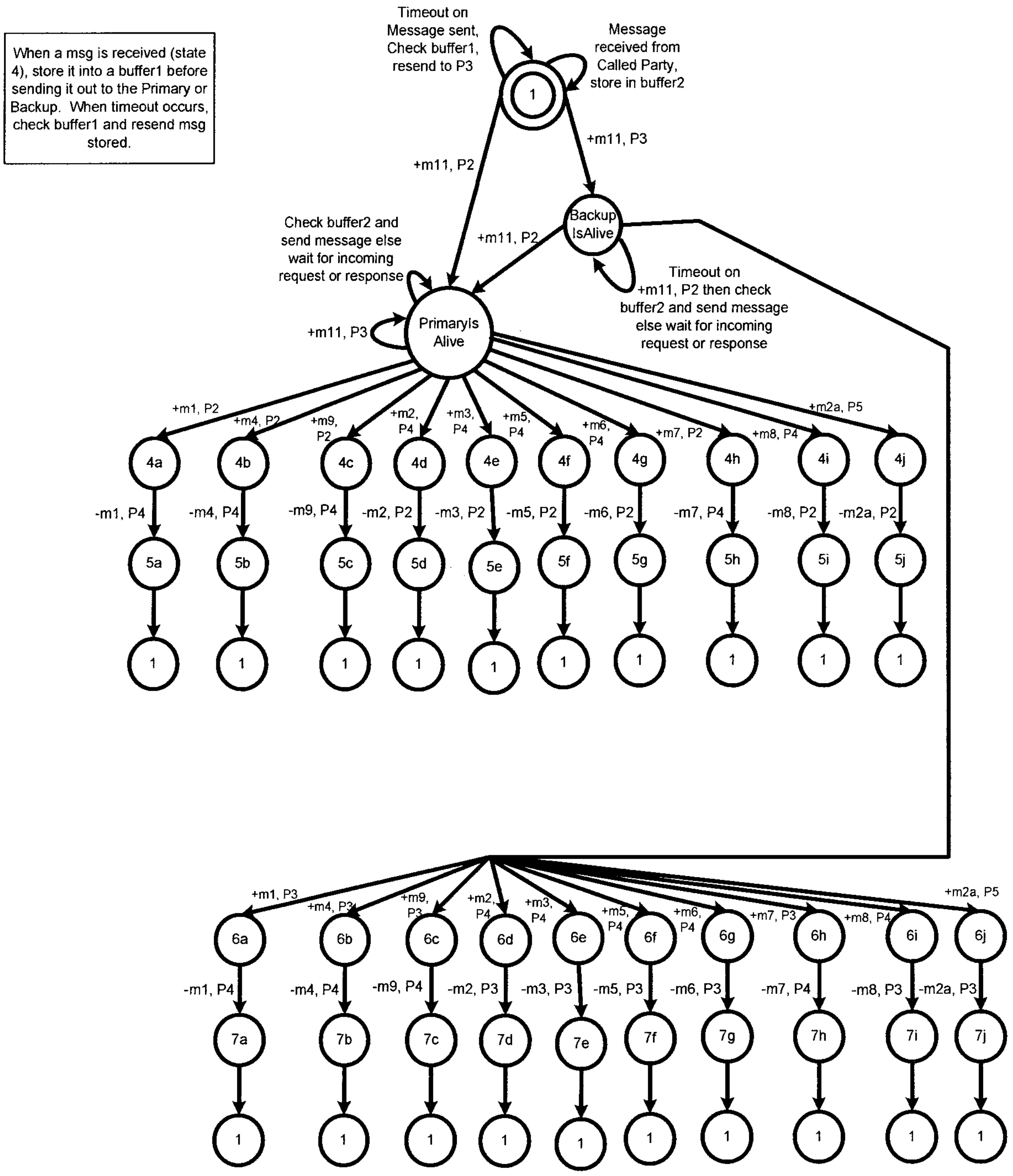

When a msg is received (state 4), store it into a buffer1 before sending it out to the Primary or Backup. When timeout occurs, check buffer1 and resend msg stored.
Timeout on Message sent, Check buffer1, resend to P3
Message received from Called Party, store in buffer2
1
+m11, P2
+m11, P3
Backup IsAlive
+m11, P2
Check buffer2 and send message else wait for incoming request or response
Timeout on +m11, P2 then check buffer2 and send message else wait for incoming request or response
+m11, P3
PrimaryIs Alive
+m1, P2
+m4, P2
+m9, P2
+m2, P4
+m3, P4
+m5, P4
+m6, P4
+m7, P2
+m8, P4
+m2a, P5
4a
4b
4c
4d
4e
4f
4g
4h
4i
4j
-m1, P4
-m4, P4
-m9, P4
-m2, P2
-m3, P2
-m5, P2
-m6, P2
-m7, P4
-m8, P2
-m2a, P2
5a
5b
5c
5d
5e
5f
5g
5h
5i
5j
+m1, P3
+m4, P3
+m9, P3
+m2, P4
+m3, P4
+m5, P4
+m6, P4
+m7, P3
+m8, P4
+m2a, P5
6a
6b
6c
6d
6e
6f
6g
6h
6i
6j
-m1, P4
-m4, P4
-m9, P4
-m2, P3
-m3, P3
-m5, P3
-m6, P3
-m7, P4
-m8, P3
-m2a, P3
7a
7b
7c
7d
7e
7f
7g
7h
7i
7j

Figure 3.7 Decider server connected to the Called Party (P6) State Machine Diagram

3.4.3 SIP Fault Scenarios

Employing the primary backup algorithm assures that one server will be actively handling the SIP Session at all times however, the location at which faults occur must be identified. We have analyzed the SIP Session both through the Session Establishment and Session Termination phases and we have determined the location at which faults can occur within the system which may affect the proper workings of the SIP Session. Furthermore, we have generalized the fault occurrences and their effect on the running session with the presence of *n* proxy servers (i.e. *n* hops) located between the end user agents.

Faults can occur within a SIP Session at the locations shown in Figure 3.8. The server can either fail before the request or response is received or else after it has been sent. At each stage, the effect on the session differs since the Decider server will have to assess the situation and retransmit the messages in order to handle the occurrence of a server fault. These effects are summarized in Table 3.3. The occurrence of all of these faults can be recovered from using our proposed approach.

Figures 3.9 to 3.15 are block diagrams which include the Decider Block solution and display the sequence of messages exchanged in order to tolerate the occurrence of server faults. Note that once the backup proxy server is used, all messages will flow along the same path until the primary proxy server recovers from the fault.

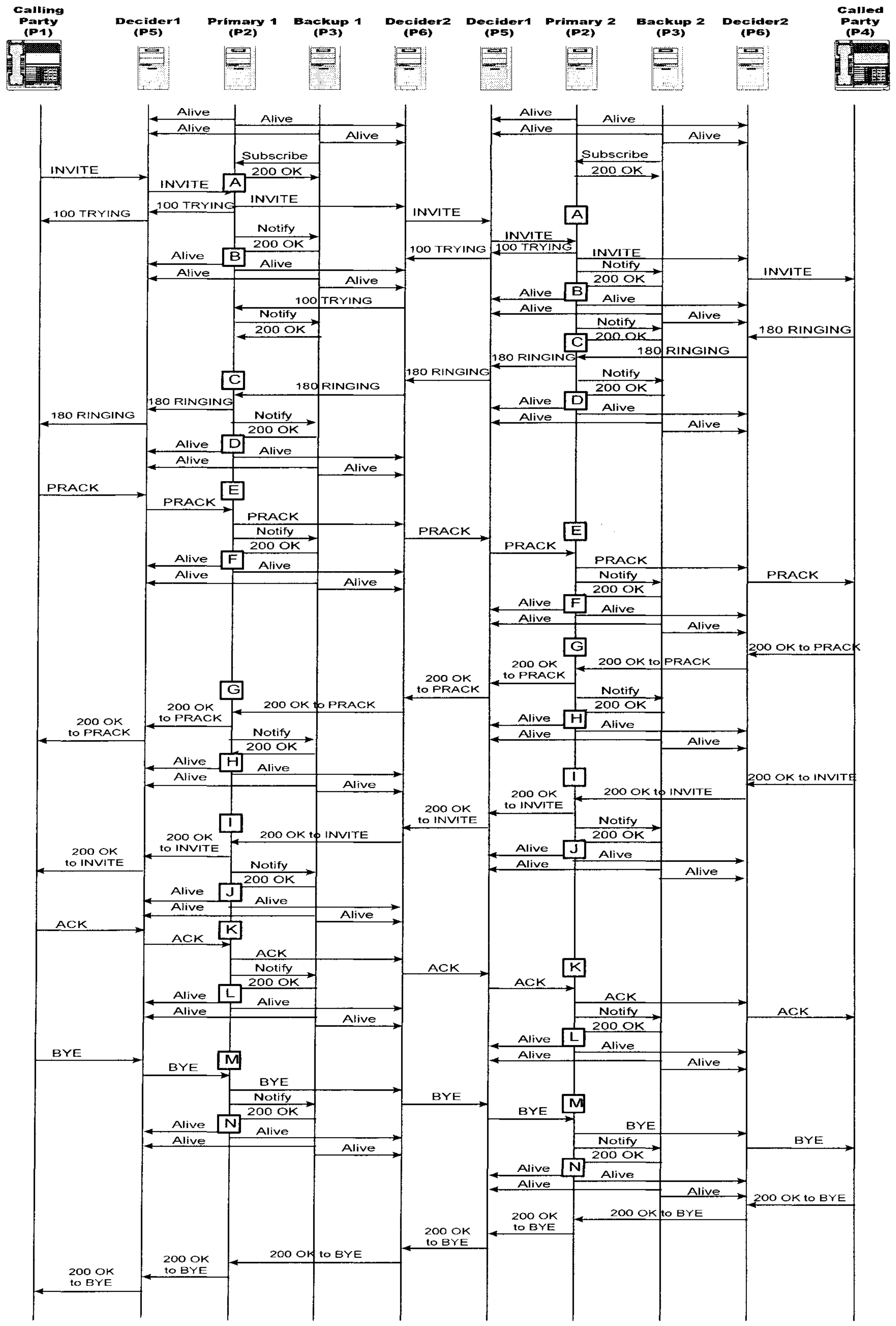


Figure 3.8 Location of fault occurrences

Server Fault Location	Effect on SIP Session
A	The Decider block (called party end) will not receive an *Alive* message from the primary proxy server therefore the INVITE will be issued to the backup proxy server. The backup will take over until the primary recovers.
B	Invite message will be sent however the Decider block (calling party end) will not receive an *Alive* message from the primary proxy server therefore the 180 RINGING will be issued to the backup proxy server. The backup will take over until the primary recovers.
C	Decider Internal Timeout Mechanism used to detect the fault. Decider block retransmits the message saved in buffer1 to the backup proxy server.
D	The Decider block (calling party end) will not receive an *Alive* message from the primary proxy server therefore the PRACK message will be sent to the backup. The backup will take over until the primary recovers.
E	Decider Internal Timeout Mechanism used to detect the fault. Decider block retransmits the message saved in buffer1 to the backup proxy server.
F	The Decider block (called party end) will not receive an *Alive* message from the primary proxy server therefore the 200 OK to PRACK message will be sent to the backup. The backup will take over until the primary recovers.
G	Decider Internal Timeout Mechanism used to detect the fault. Decider block retransmits the message saved in buffer1 to the backup proxy server.
H	The Decider block (called party end) will not receive an *Alive* message from the primary proxy server therefore the 200 OK to INVITE message will be sent to the backup. The backup will take over until the primary recovers.
I	Decider Internal Timeout Mechanism used to detect the fault. Decider block retransmits the message saved in buffer1 to the backup proxy server.
J	The Decider block (calling party end) will not receive an *Alive* message from the primary proxy server therefore the ACK message will be sent to the backup. The backup will take over until the primary recovers.
K	Decider Internal Timeout Mechanism used to detect the fault. Decider block retransmits the message saved in buffer1 to the backup proxy server.
L	The Session is already established so media flows directly between end users. Proxy server fault has no effect.
M	The Decider block (calling party end) will not receive an *Alive* message from the primary proxy server therefore the BYE message will be sent to the backup. The backup will take over until the primary recovers. If no 200 OK to Bye is sent, a timeout occurs and all resources are de-allocated and the session is terminated by all SIP entities.
N	The Calling Party will terminate (even without receipt of the 200 OK to BYE). No media will be sent so after a timeout, the Called party will also terminate its session connection.

Table 3.3 Effect of fault on SIP session

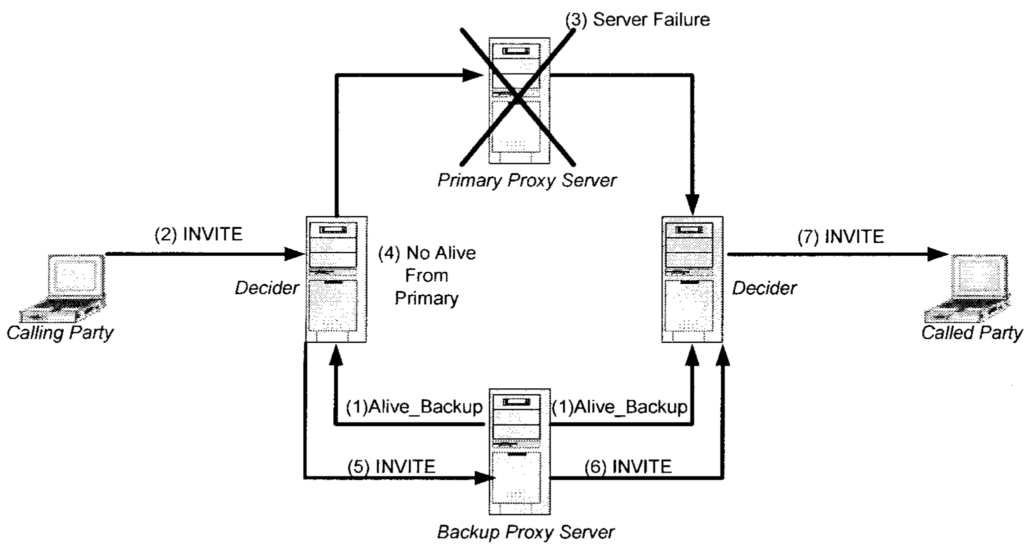


Figure 3.9 Fault A – Proxy fails before receiving INVITE

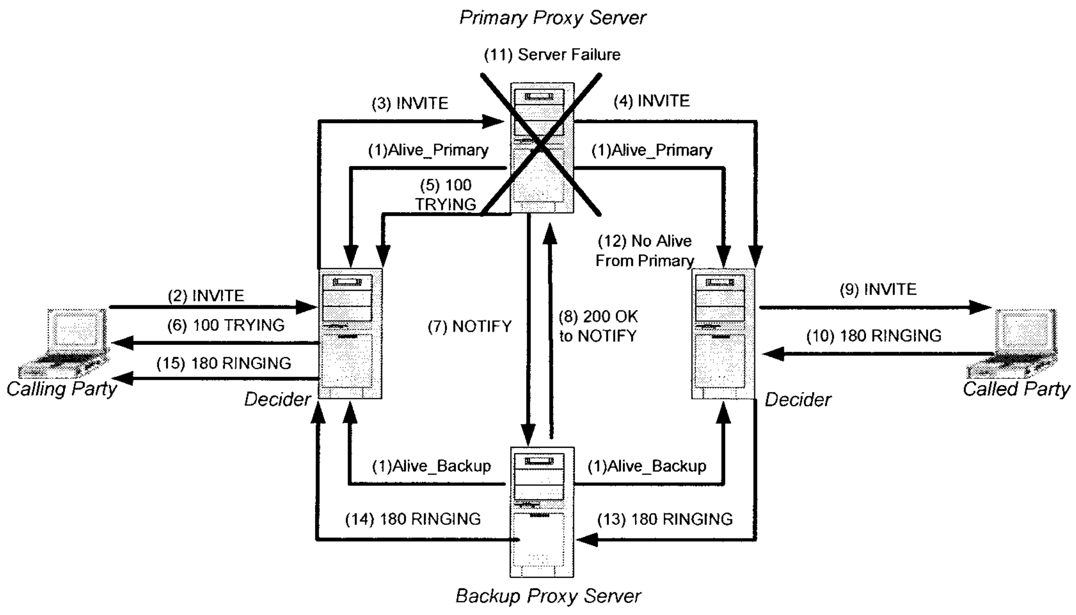


Figure 3.10 Fault B – Proxy fails after sending 100 TRYING

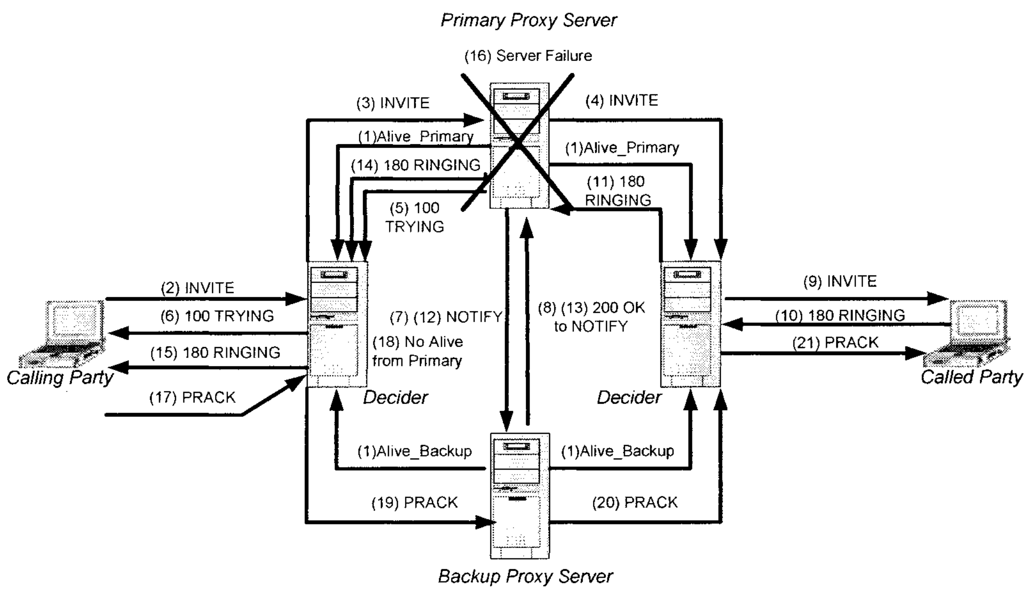


Figure 3.11 Fault D – Proxy fails after sending 180 Ringing or Error Response

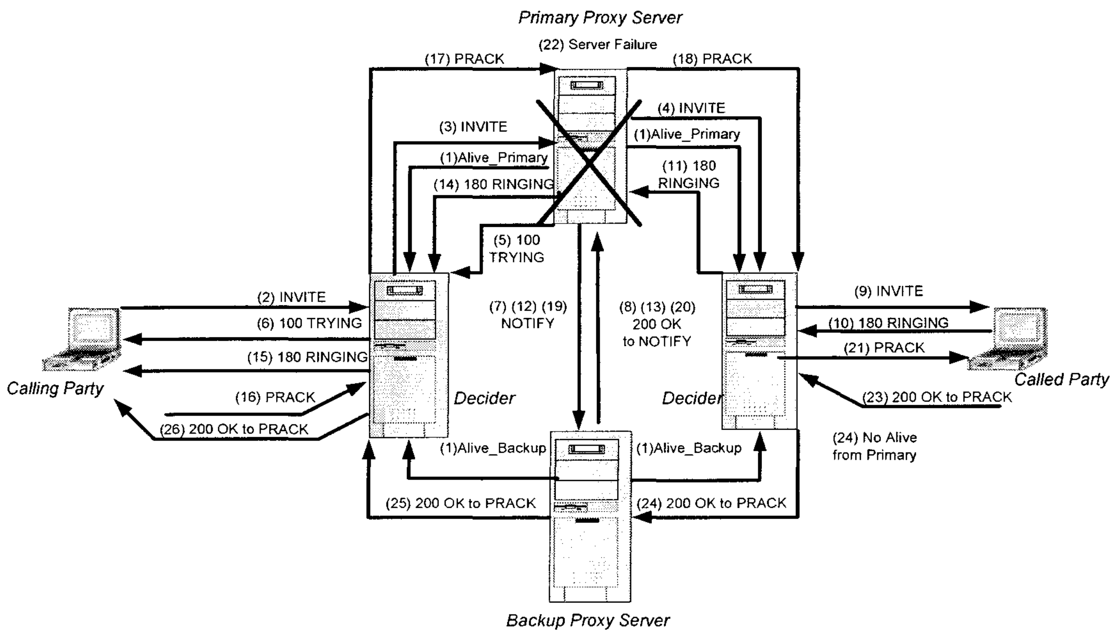


Figure 3.12 Fault F – Proxy fails after sending PRACK

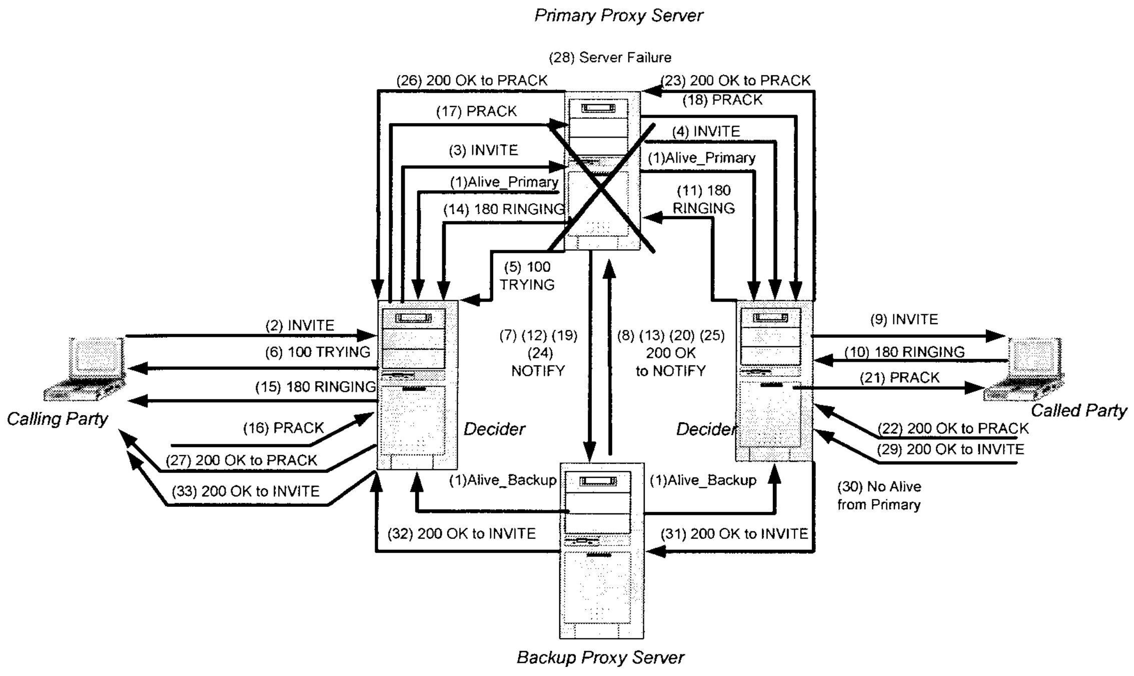


Figure 3.13 Fault H – Proxy fails after sending 200 OK to PRACK

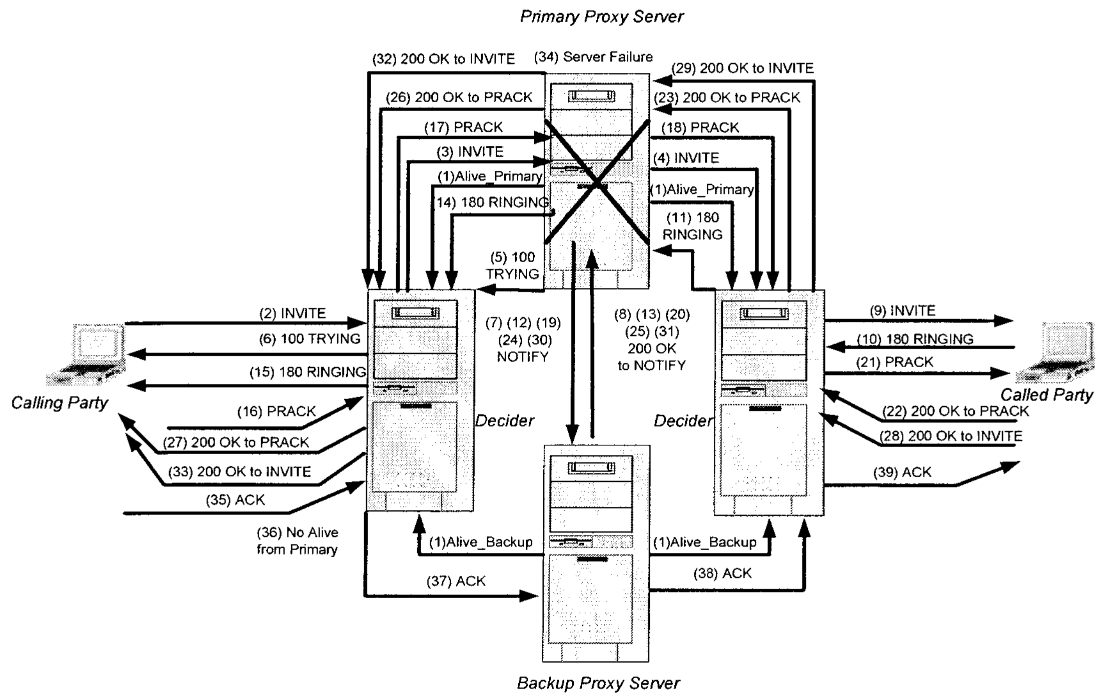


Figure 3.14 Fault J – Proxy fails after sending 200 OK to INVITE

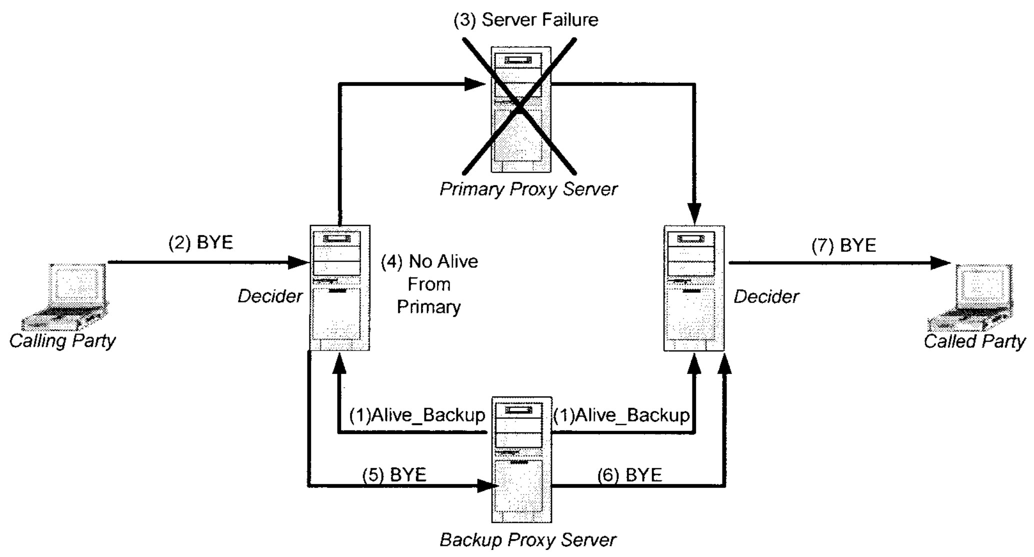


Figure 3.15 Fault M – Proxy fails before receiving BYE

Figure 3.16 is the only situation which uses the timeout mechanism. This situation arises when the primary proxy server fails after sending an *Alive* message to the Decider block. In this case, the Decider will identify the fault through an internal timeout and it will resend the message to the backup proxy server. This can occur at any point within the SIP Session however the end users are hidden from this retransmission.

At any point within the SIP Session, the primary proxy server can recover from the fault which had occurred. In order to resume its responsibility within the session, it must request the current state from the backup proxy server so that it remains consistent with the running session. Once the state of the session is retrieved, the primary proxy server sends an *Alive* message to inform the Decider server of its restoration. As a result, the messages flow through the primary proxy server until another server fault occurs. An example of this situation is shown is Figure 3.17. In this case the server fails after the INVITE request has been sent to the primary proxy server.

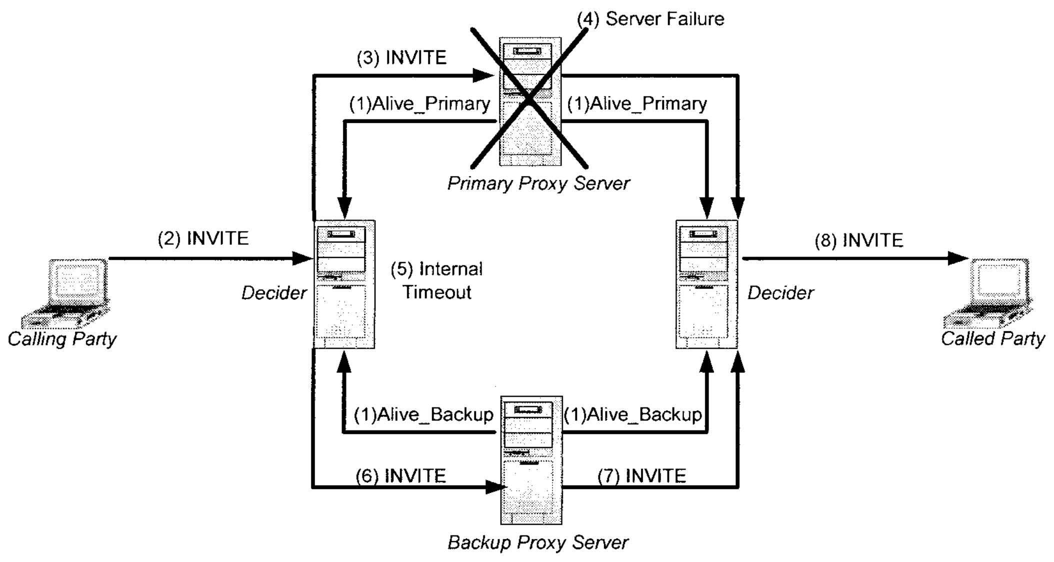


Figure 3.16 Fault C, E, G, I, K – Proxy fails after receiving INVITE

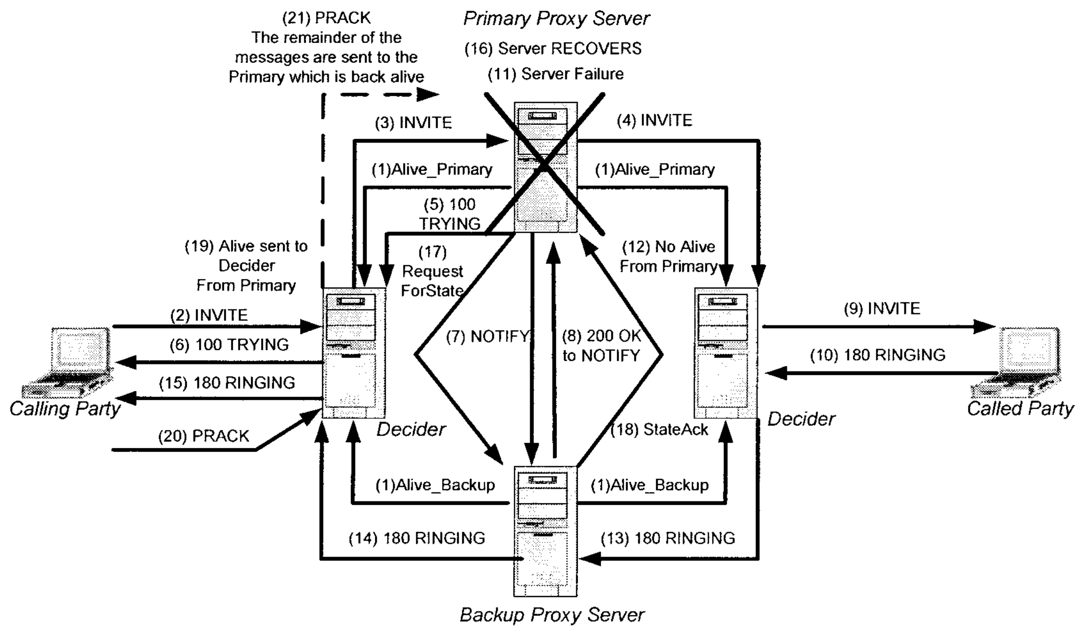


Figure 3.17 Primary proxy recovers from fault and resumes operation

3.5 Summary

This chapter begins with the identification of the link and proxy failures which can occur within a SIP session. Mechanisms to handle these failures within the system and determination of solutions which incur small amounts of delay are the focus of our proposed approach. The chapter then provides an overview of the algorithms which have been suggested to achieve Fault Tolerance within SIP. It further discusses the shortcomings of these approaches which serve as a basis for our work. We then discuss our proposed approach and introduce a Decider server. This solution is advantageous for two reasons. First it eliminates the retransmission of messages from the end user and secondly it considerably reduces the dependency on timeout mechanisms which generally delays the overall SIP session. We then identify the locations within a SIP session where faults can occur and affect on the message flow. We conclude the chapter with a diagrammatic description of messages exchanged between SIP elements including the Decider server, in the presence of server faults.

Chapter 4 Implementation

In this chapter, we introduce the tool used to simulate our proposed fault tolerant block within SIP. We provide an overview of the capabilities of the tool, the components within its system and the results which can be achieved from this simulation. We then illustrate the design of SIP within the tool. This includes all of the SIP components including the Decider server. We further describe the finite state machines which represent the message flow which each entity traverses. This leads to the analysis of the timing constraints of the system as well as the theoretical performance factors within the system. We conclude with the assumptions taken when designing the system.

4.1 SDL Tool

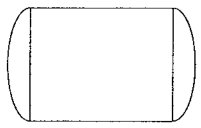
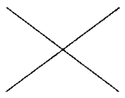
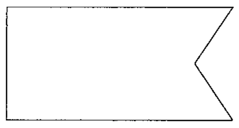
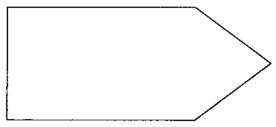
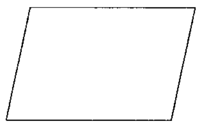
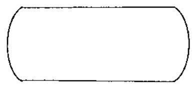
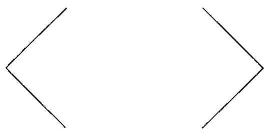
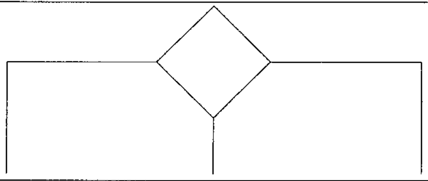
The tool used for simulation of our Fault Tolerant SIP solution is Specification and Description Language (SDL) [22]. SDL is a standardized language used for the description of systems as communicating state machines. It is widely used in the area of telecommunications and protocol systems however it is currently expanding into other areas such as aircraft and railway systems. The key features of this tool are:

- Suitable for communicating distributed systems
- Offers Real-time capabilities
- Provides both graphical and textual representations of the system
- Offers the creation of a Stimulus-Response paradigm
- Provide a solid platform for developers and testers

Since SDL is a message based system, communication protocols can be implemented and then checked for correctness based on the Message Sequence Charts (MSC) derived from the tool. The basic notion is that the entities of the protocol are represented as an Extended Finite State Machine (EFSM) and each one can exchange messages through queues. These queues follow the FIFO (First-in-First-out) algorithm wherein the messages will be processed in the order that they are received.

There are four main hierarchical levels in SDL. The first is the System level which is the uppermost construct and represents the overall structure of the system to be designed in SDL. Within this level, the communication between entities is shown and moreover, the system's response to environmental stimuli is described. This system is then composed of many Blocks which is the second level within the hierarchy. These blocks are individual entities and may be subdivided into sub blocks. Both the System and Block levels have channels which serve as the medium over which communication can take place. The blocks contain Processes which is the third hierarchy level. A Process in SDL is essentially an EFSM which describes the workings of the stated block. It contains the states that the block can reach, the variable types that the block can handle and moreover any timing constraints which are present. Furthermore, the process terminates once its execution has completed. The final hierarchy level is the Procedure which can be placed within any of the previous levels and describes a temporary finite state machine. It is local to the level in which it is created and can repeat execution from the initial state.

There are several important constructs that are available in SDL. They are used to create the EFSM within the process level and are shown in Table 4.1. These allow the descriptions of the internal workings of the system. These EFSMs can run concurrently and express the reaction to stimuli. They are either stagnant at a particular state or in the middle of the execution of a transition.

Symbol	**Description**
	Start
	Stop
	Input
	Output
	Save
	State
	Continuous Signal
	Decision

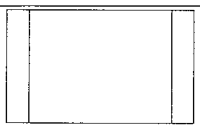
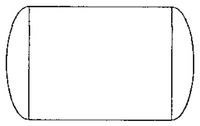
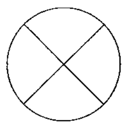
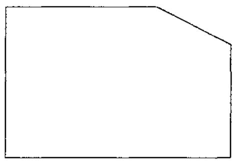
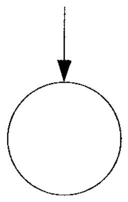
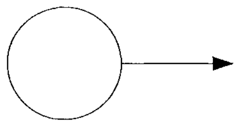
	Procedure Call
	Procedure Start
	Procedure Return
	Text Box
	Join
	Connection/Label

Table 4.1 Symbols used in SDL to represent FSM entities

There are also several data types which are considered within SDL. These can be used when declaring variables within the EFSM. These include Boolean (True or False), Character ('A', 'B', etc.), Integer (1, 2, 45, etc.), Natural (Null or Positive Integer), Real (43.7, etc.) and Time (used for the creation of Timers).

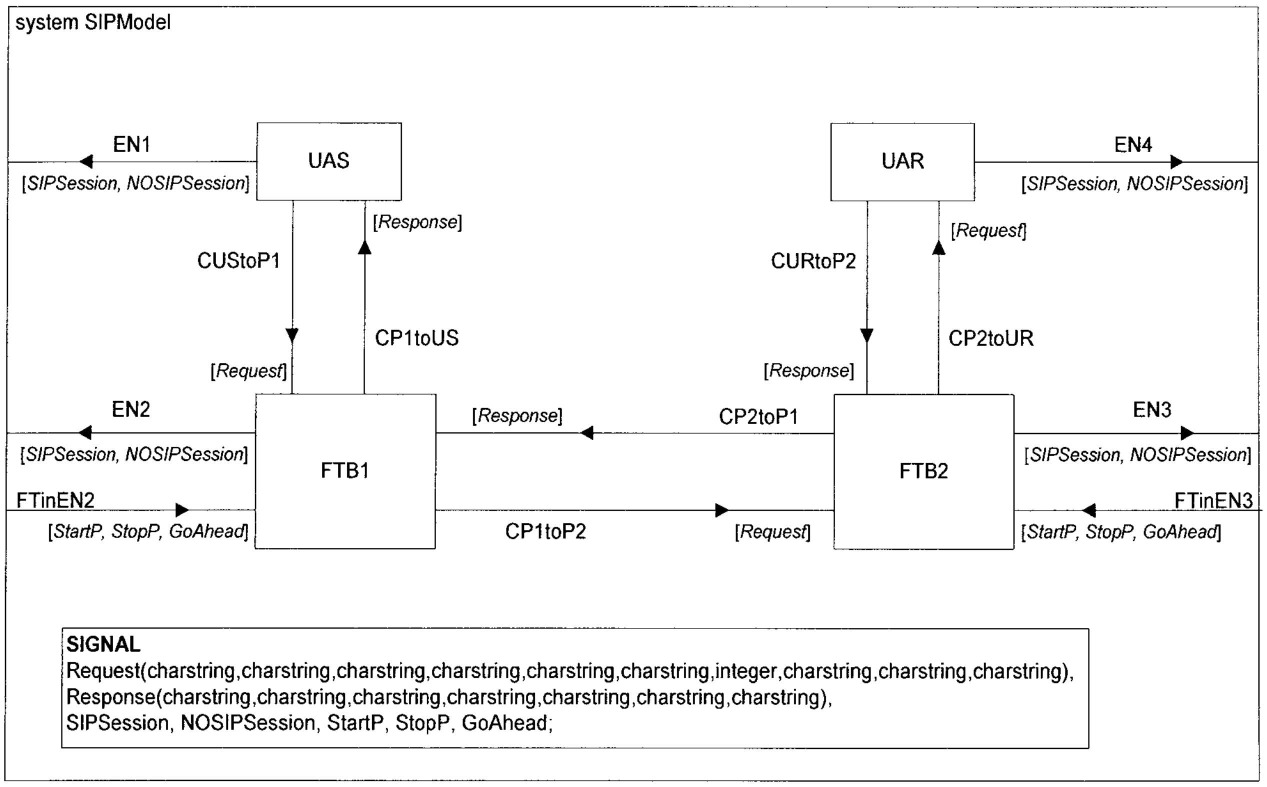
The constructs and data types available within SDL allow systems to be defined as EFSM. Through these, communication protocols can be verified to see if they are following the expected set of rules and can also be validated for their correctness with respect to the desired workings of the protocol.

4.2 System Design

We have implemented our proposed approach including the Decider server within SDL [23]. The System level of the model is shown in Figure 4.1. The system is composed of four blocks representing the entities communicating within the SIP session. *UAS* represents the Calling Party which instantiates the initial INVITE message. *UAR* is the Called Party which issues responses based on whether it wishes to join the session or not. The two middle blocks *FTB1* and *FTB2* are the fault tolerance blocks which we have proposed. They contain the functionality to handle server failures which may occur without the system. The managing of faults is internal to the fault tolerance blocks and thus the workings are not visible to the end users *UAS* and *UAR*.

The status of the session (established or not) is informed to the environment through channels *EN1*, *EN2*, *EN3*, and *EN4*. These notify that a SIP Session has been established between the end users (*SIPSession*) or that an error has occurred and no session has started (*NOSIPSession*).

The channels between the blocks (*CUStoP1*, *CP2toUR* etc.) allow the requests and responses to traverse through the system from end to end. For clarity within this implementation, the requests traverse from the *UAS* to the *UAR* block whereas the responses in the opposite direction. Finally, there are three messages which can be inputted from the environment. These are *StartP*, *StopP*, and *GoAhead.* These messages are used to feed faults within the system. *StopP* will disable the primary proxy server emulating that a fault has occurred. *StartP* will enable the primary proxy server emulating that it has recovered from the previously occurred fault. *GoAhead* is used to emulate that there is no fault within the system and thus the Decider server can be informed so that it sends messages through the primary proxy server.



SIGNAL
Request(charstring,charstring,charstring,charstring,charstring,charstring,integer,charstring,charstring,charstring),
Response(charstring,charstring,charstring,charstring,charstring,charstring,charstring),
SIPSession, NOSIPSession, StartP, StopP, GoAhead;

Figure 4.1 Overall SDL System Model

4.3 Block and Procedure Design

The Blocks within the model contain the process to be described as an EFSM as well as the channels along which messages enter and exit the block. The first is the *UAS* Block as shown in Figure 4.2. This contains the *SIP_Sender* process which will define the workings of the Calling Party within the SIP Session. Furthermore this block contains several processes which create an instance of a request. As described in Chapter 2, the SIP User Agent contains an embedded transaction user which creates messages. This layer is represented in our model using several processes which create a request and then terminate. These processes are executed only once as there is no retransmission of requests with a SIP session. The processes included in the *UAS* block are for the following requests; INVITE (*TU_INV*) shown in Figure 4.3, ACK (*TU_ACK*) shown in Figure 4.4, PRACK (*TU_PRACK*) shown in Figure 4.5, and BYE (*TU_BYE*) shown in Figure 4.6. The setup for all of the requests is similar however the contents of the message differ. The process begins, creates the request and outputs the message to the Sender block and then terminates.

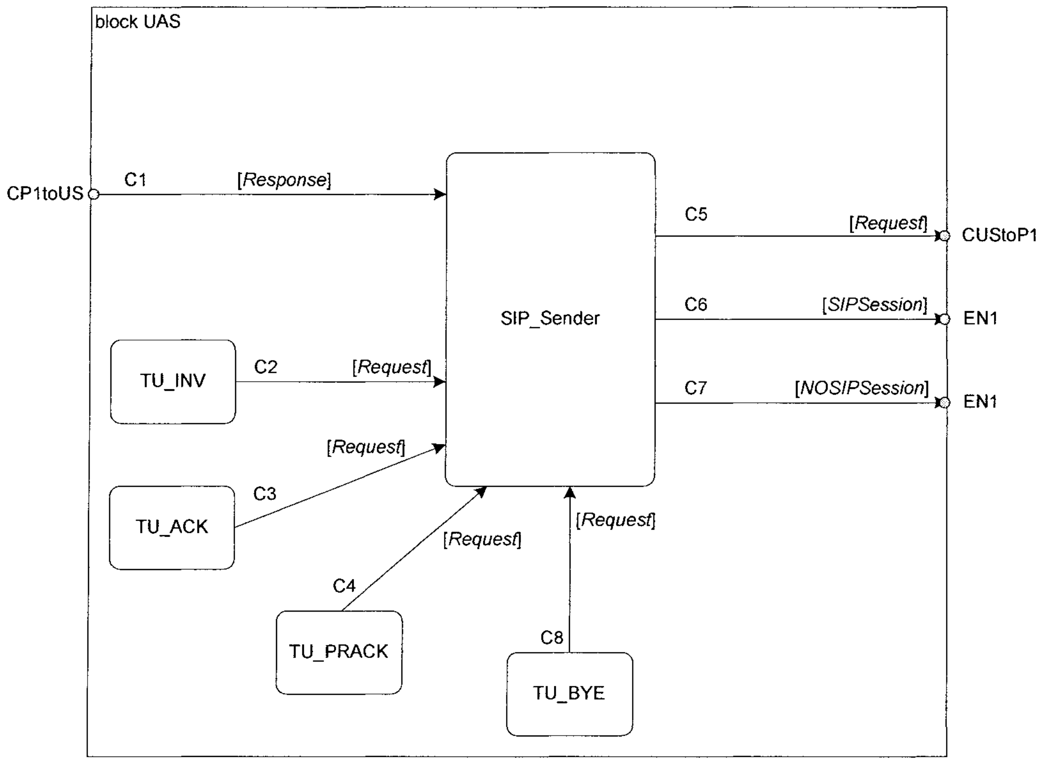


Figure 4.2 Block Model of Calling Party

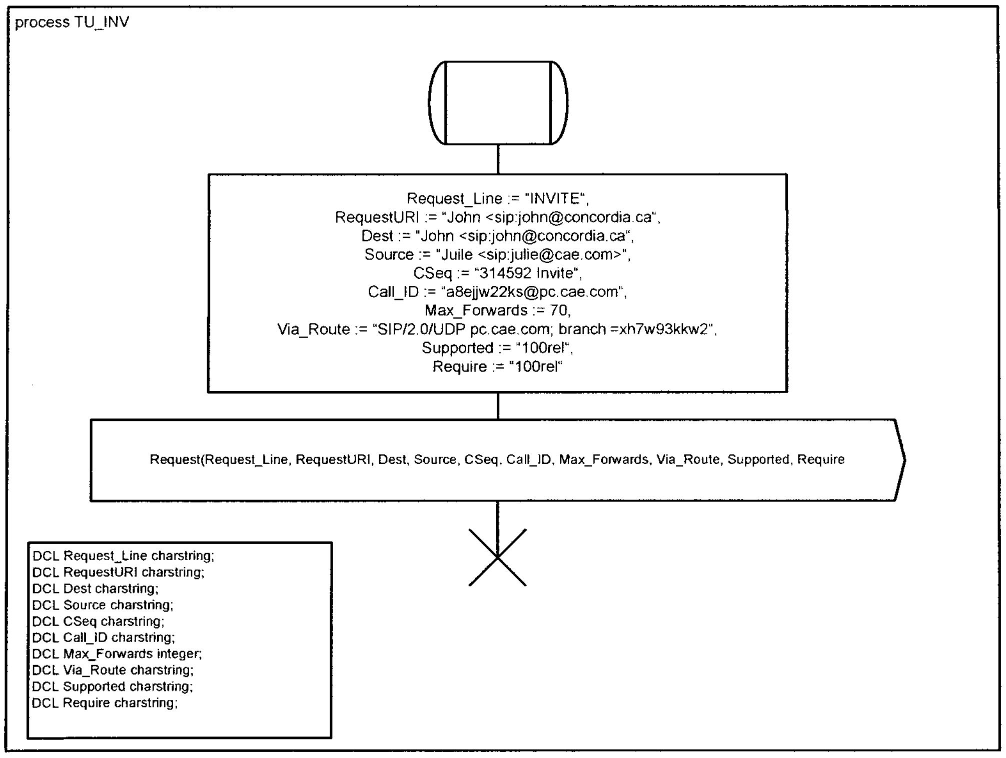

process TU_INV
Request_Line := "INVITE",
RequestURI := "John <sip:john@concordia.ca",
Dest := "John <sip:john@concordia.ca",
Source := "Juile <sip:julie@cae.com>",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Max_Forwards := 70,
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2",
Supported := "100rel",
Require := "100rel"
Request(Request_Line, RequestURI, Dest, Source, CSeq, Call_ID, Max_Forwards, Via_Route, Supported, Require
DCL Request_Line charstring;
DCL RequestURI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Max_Forwards integer;
DCL Via_Route charstring;
DCL Supported charstring;
DCL Require charstring;

Figure 4.3 Creation of INVITE Request

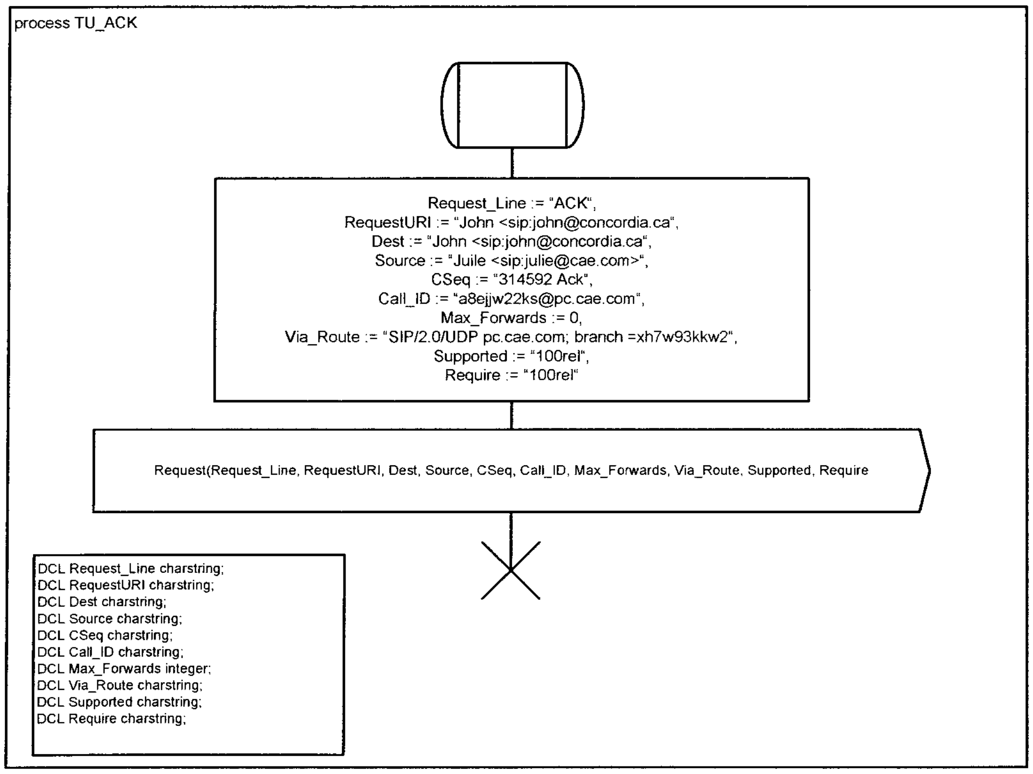

process TU_ACK
Request_Line := "ACK",
RequestURI := "John <sip:john@concordia.ca",
Dest := "John <sip:john@concordia.ca",
Source := "Juile <sip:julie@cae.com>",
CSeq := "314592 Ack",
Call_ID := "a8ejjw22ks@pc.cae.com",
Max_Forwards := 0,
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2",
Supported := "100rel",
Require := "100rel"
Request(Request_Line, RequestURI, Dest, Source, CSeq, Call_ID, Max_Forwards, Via_Route, Supported, Require
DCL Request_Line charstring;
DCL RequestURI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Max_Forwards integer;
DCL Via_Route charstring;
DCL Supported charstring;
DCL Require charstring;

Figure 4.4 Creation of ACK Request

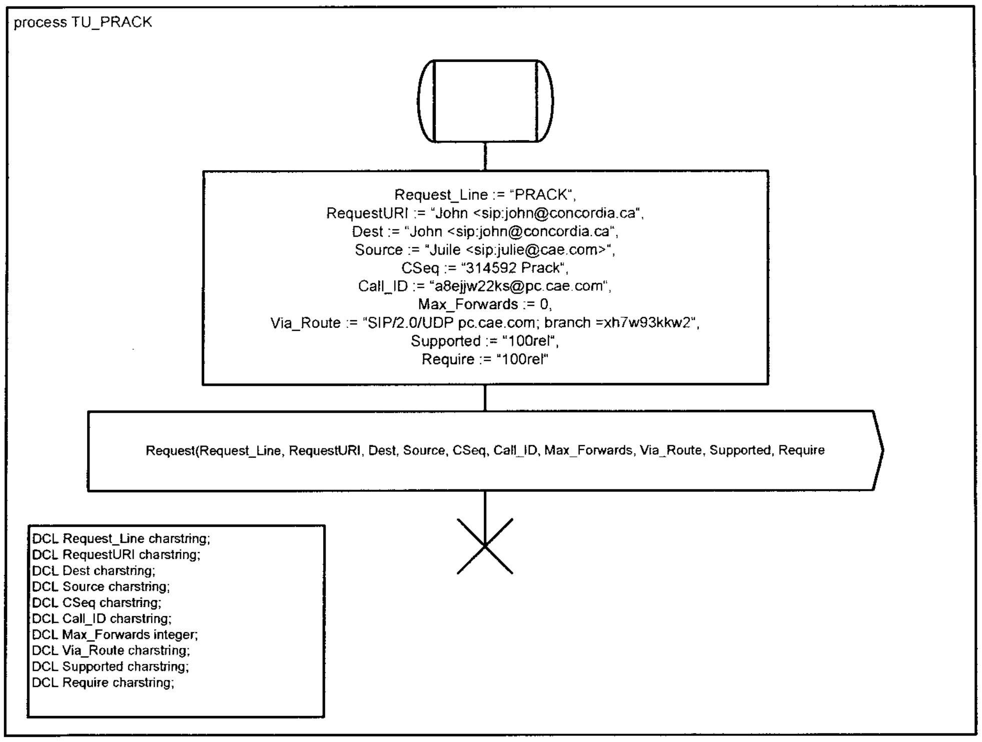

process TU_PRACK
Request_Line := "PRACK",
RequestURI := "John <sip:john@concordia.ca",
Dest := "John <sip:john@concordia.ca",
Source := "Juile <sip:julie@cae.com>",
CSeq := "314592 Prack",
Call_ID := "a8ejjw22ks@pc.cae.com",
Max_Forwards := 0,
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2",
Supported := "100rel",
Require := "100rel"
Request(Request_Line, RequestURI, Dest, Source, CSeq, Call_ID, Max_Forwards, Via_Route, Supported, Require
DCL Request_Line charstring;
DCL RequestURI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Max_Forwards integer;
DCL Via_Route charstring;
DCL Supported charstring;
DCL Require charstring;

Figure 4.5 Creation of PRACK Request

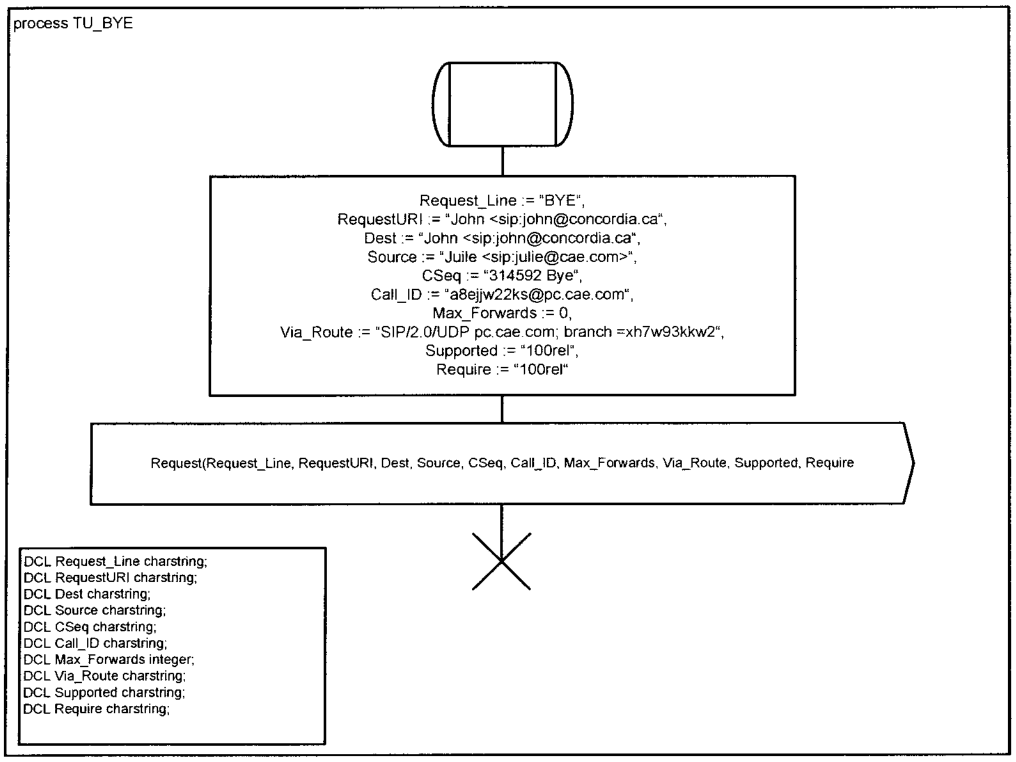

process TU_BYE
Request_Line := "BYE",
RequestURI := "John <sip:john@concordia.ca",
Dest := "John <sip:john@concordia.ca",
Source := "Juile <sip:julie@cae.com>",
CSeq := "314592 Bye",
Call_ID := "a8ejjw22ks@pc.cae.com",
Max_Forwards := 0,
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2",
Supported := "100rel",
Require := "100rel"
Request(Request_Line, RequestURI, Dest, Source, CSeq, Call_ID, Max_Forwards, Via_Route, Supported, Require
DCL Request_Line charstring;
DCL RequestURI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Max_Forwards integer;
DCL Via_Route charstring;
DCL Supported charstring;
DCL Require charstring;

Figure 4.6 Creation of BYE Request

The next is the *UAR* Block as shown in Figure 4.7. This contains the *SIP_Receiver* process which will define the workings of the Called Party within the SIP Session. Similar to the sender block, this block also contains several processes however in this case they create instances of responses. The same rules for the process apply here wherein there is no retransmission. The provisional, error and acceptance responses can be retransmitted as in the definition of SIP however this burden has been shifted to the fault tolerant block. As per the user's point of view, there are no retransmissions.

The processes included in the *UAR* block are for the following requests; PROVISIONAL RESPONSE (*TU_Prov*) shown in Figure 4.8, ERROR RESPONSE (*TU_Err*) shown in Figure 4.9, 200 OK to INVITE (*TU_Accept*) shown in Figure 4.10, 200 OK to PRACK (*TU_PrackAccept*) shown in Figure 4.11 and 200 OK to BYE (*TU_ByeAccept*) shown in Figure 4.12. The setup for all of these responses is similar however the contents of the message differ. The process begins, creates the response and outputs the message to the Receiver block.

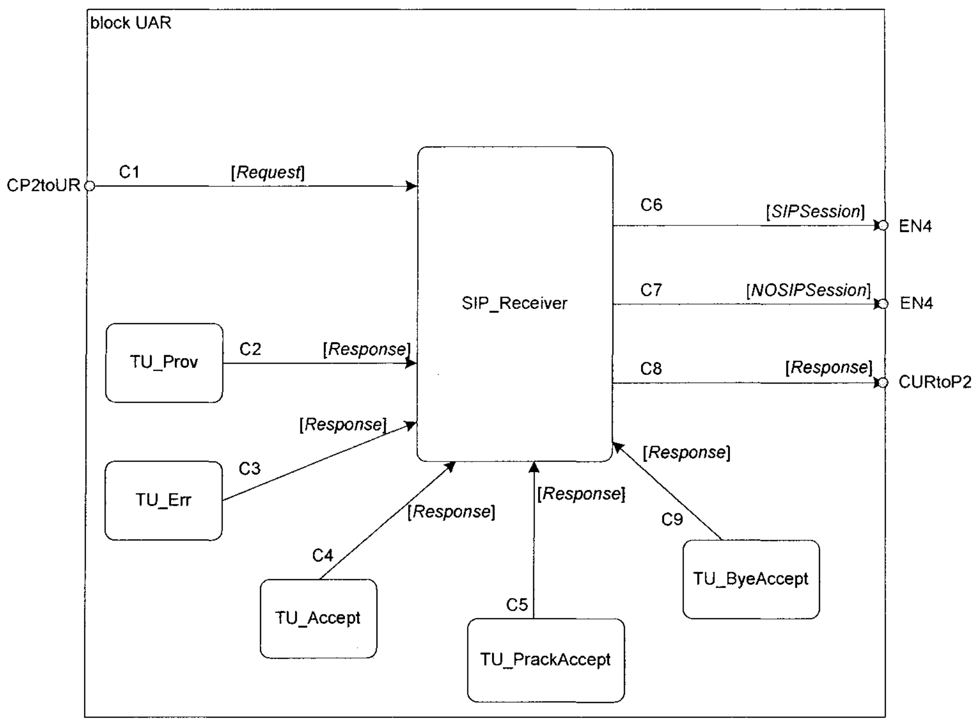
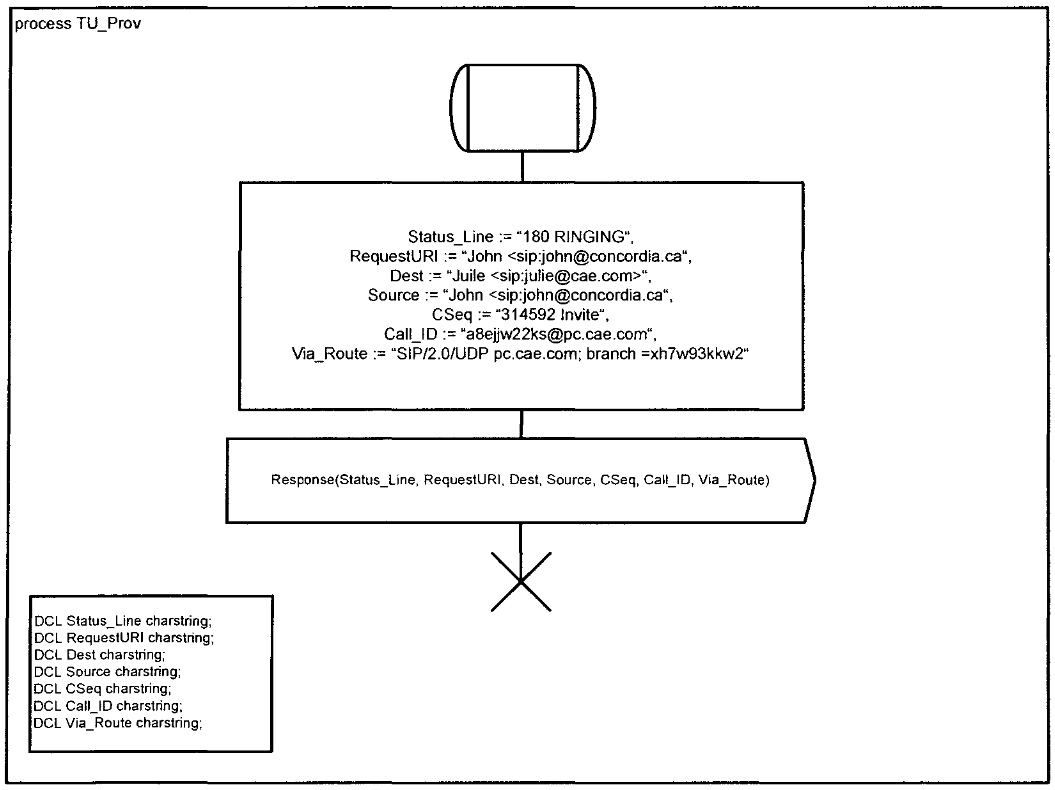


Figure 4.7 Block Model of Called Party



```
Status_Line := "180 RINGING",
RequestURI := "John <sip:john@concordia.ca",
Dest := "Juile <sip:julie@cae.com>",
Source := "John <sip:john@concordia.ca",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2"
```

```
Response(Status_Line, RequestURI, Dest, Source, CSeq, Call_ID, Via_Route)
```

```
DCL Status_Line charstring;
DCL RequestURI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;
```

Figure 4.8 Creation of 180 RINGING Response

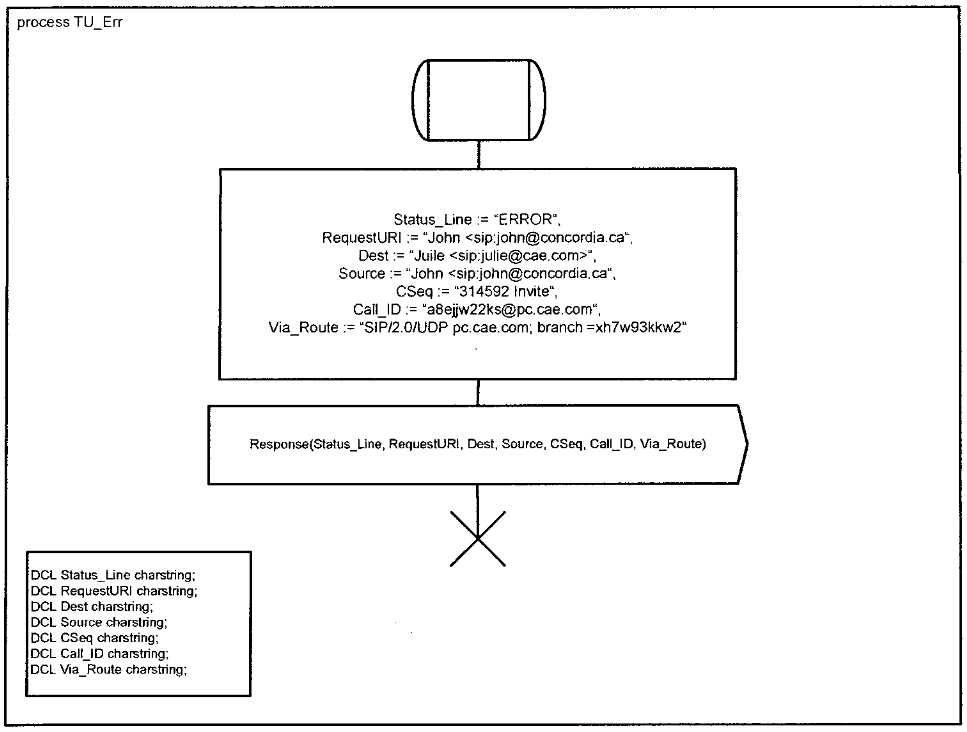

process TU_Err
Status_Line := "ERROR",
RequestURI := "John <sip:john@concordia.ca",
Dest := "Juile <sip:julie@cae.com>",
Source := "John <sip:john@concordia.ca",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2"
Response(Status_Line, RequestURI, Dest, Source, CSeq, Call_ID, Via_Route)
DCL Status_Line charstring;
DCL RequestURI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;

Figure 4.9 Creation of ERROR Response

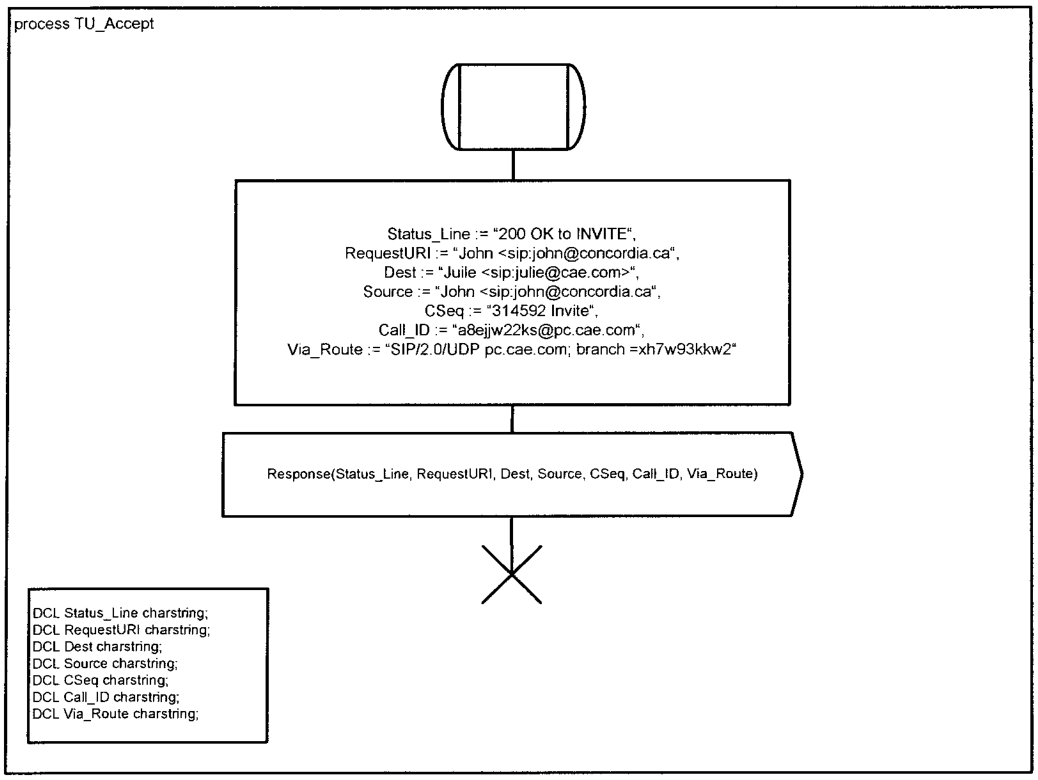

process TU_Accept
Status_Line := "200 OK to INVITE",
RequestURI := "John <sip:john@concordia.ca",
Dest := "Juile <sip:julie@cae.com>",
Source := "John <sip:john@concordia.ca",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2"
Response(Status_Line, RequestURI, Dest, Source, CSeq, Call_ID, Via_Route)
DCL Status_Line charstring;
DCL RequestURI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;

Figure 4.10 Creation of 200 OK to INVITE Response

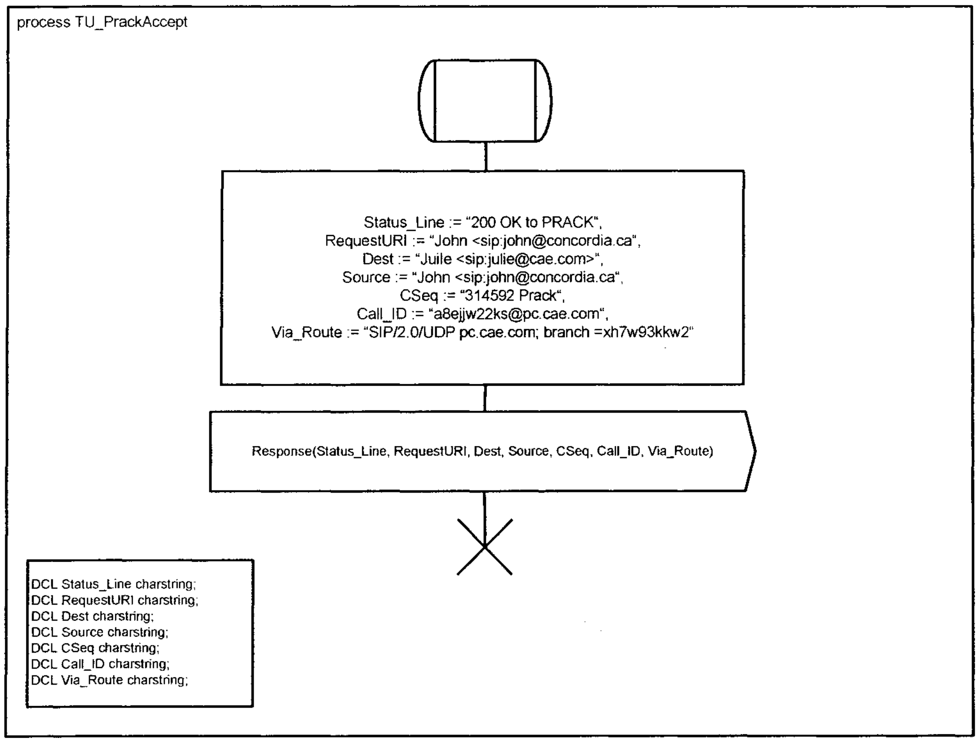

process TU_PrackAccept
Status_Line := "200 OK to PRACK",
RequestURI := "John <sip:john@concordia.ca",
Dest := "Juile <sip:julie@cae.com>",
Source := "John <sip:john@concordia.ca",
CSeq := "314592 Prack",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2"
Response(Status_Line, RequestURI, Dest, Source, CSeq, Call_ID, Via_Route)
DCL Status_Line charstring;
DCL RequestURI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;

Figure 4.11 Creation of 200 OK to PRACK Response

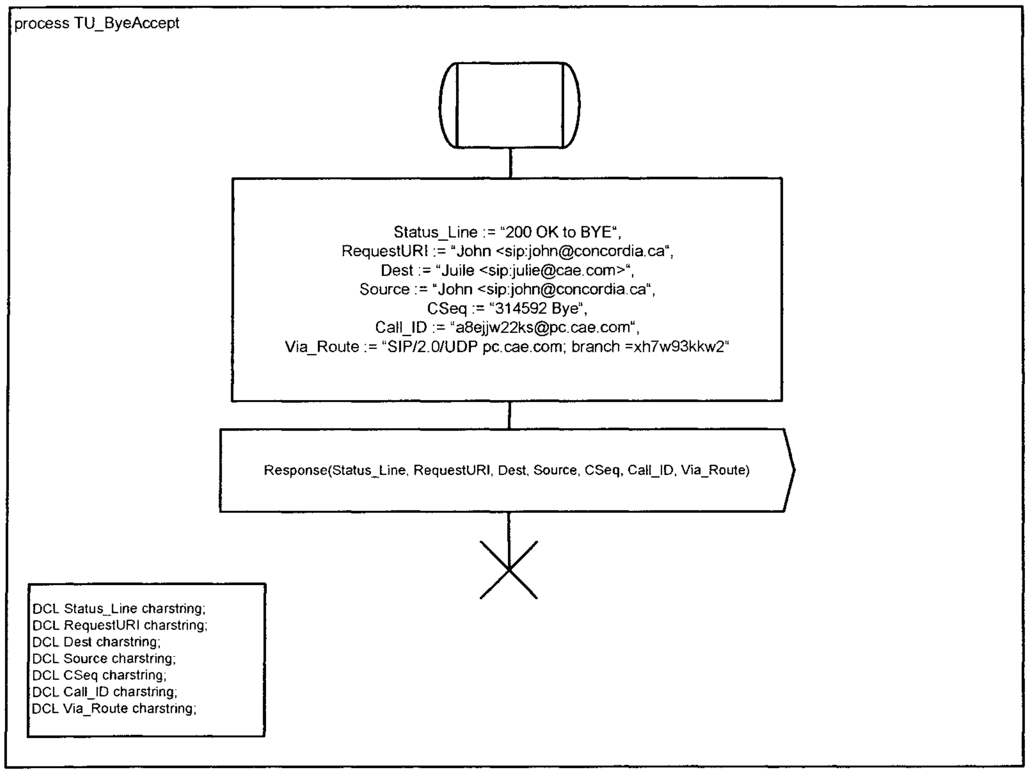

process TU_ByeAccept
Status_Line := "200 OK to BYE",
RequestURI := "John <sip:john@concordia.ca",
Dest := "Juile <sip:julie@cae.com>",
Source := "John <sip:john@concordia.ca",
CSeq := "314592 Bye",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2"
Response(Status_Line, RequestURI, Dest, Source, CSeq, Call_ID, Via_Route)
DCL Status_Line charstring;
DCL RequestURI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;

Figure 4.12 Creation of 200 OK to BYE Response

The next two blocks are similar since they represent two proxies existing between the end user agents. Each fault tolerant block contains a set of four sub blocks as shown in Figure 4.13. These are *Decider1* which is the Decider server connected to the Calling Party, *Primary* which is the primary proxy server within the domain, *Backup* which is the backup proxy server within the domain and *Decider2* which is the Decider server connected to the Called Party. The channels local to this block are the *AliveP* and *AliveB* messages which are sent to the Decider server from the primary and backup Proxies respectively. This informs the Decider server about the availability of the proxy server. The other local messages are those between the primary and backup servers. These include *Subscribe*, *Notify*, *AcceptRespN* (200 OK to Notfiy), *AcceptRespS* (200 OK to Subscribe), *StateReq* wherein the primary proxy informs the backup that it has recovered and is ready to resume within the SIP Session and *StateResp* which contains the current state from the backup thereby allowing the primary to return to the session after recovery.

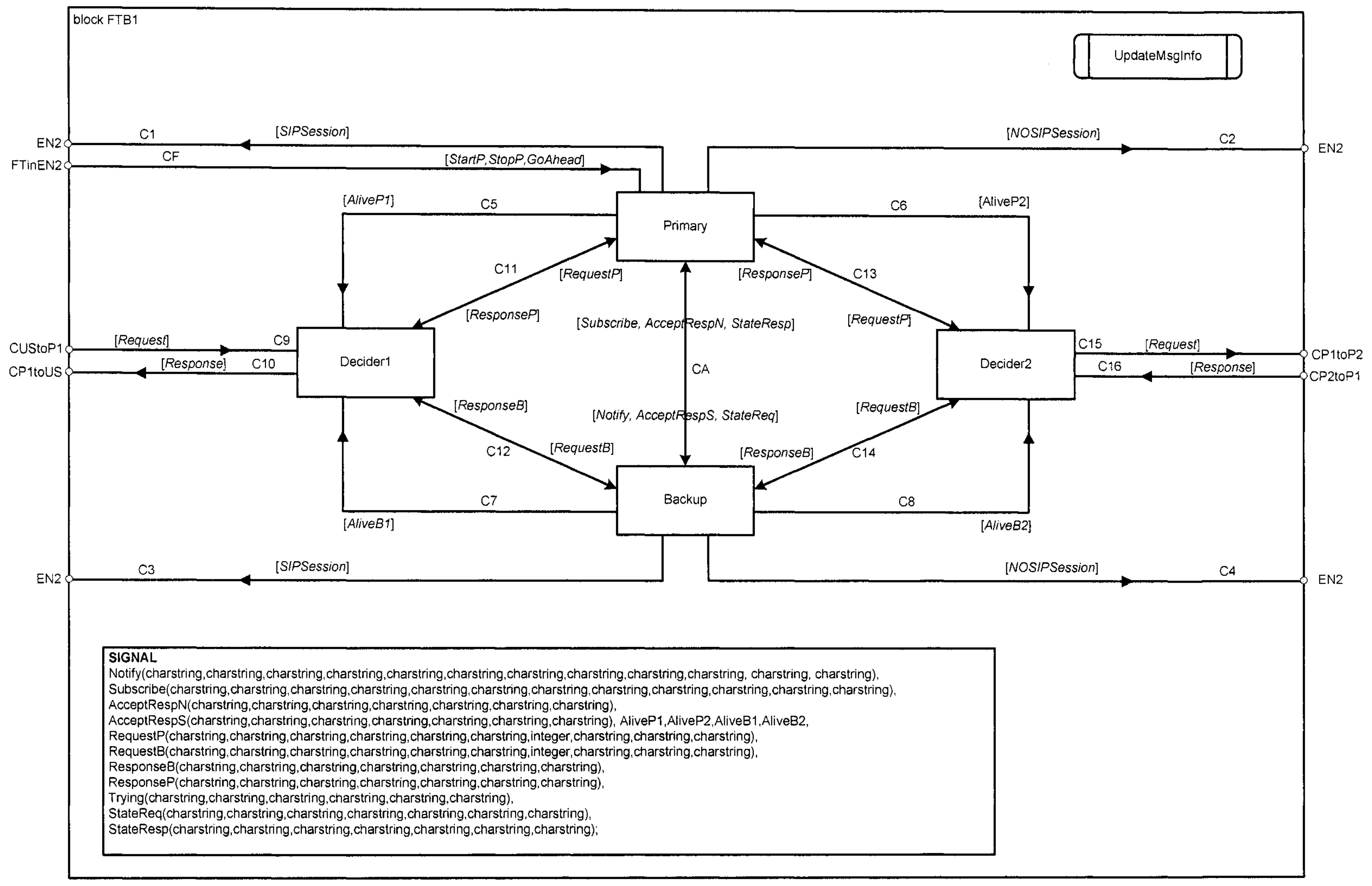
This block also contains a procedure called UpdateMsgInfo as shown in Figure 4.14 which is used to update the header parameters of the requests and responses which traverse through the proxy servers. According to the SIP rules [4], there are three changes which the proxy must make. It should,

1) Reduce the number of hops (MaxForwards header field) in each request sent
2) Add the location which it wishes to receive messages at (Via header field)
3) Update the route, if required (RequestURI header field)

The Decider sub blocks as shown in Figures 4.15 and 4.16 are similar except that they are connected to the Calling or Called parties respectively. They contain the processes *D_Process1* and *D_Process2* and the channels along which the requests and responses are sent. The *AliveP1* message informs the Decider1 that the primary proxy server is alive and able to receive requests. The *AliveP2* message informs the Decider2 that the primary proxy server is alive and able to receive responses. The same goes for the *AliveB1* and *AliveB2* messages except they inform of the availability of the backup server.

The next sub block is the primary proxy block shown in Figure 4.17. This contains the *Primary_Process* process which defines the workings of the primary proxy server. The block contains the channels of messages that flow through the proxy. There are two processes within this block. The first is *TU_Trying* shown in Figure 4.18 which creates the 100 TRYING message and sends it to the previous hop. This message as described in Chapter 2 is hop by hop and thus locally generated. The second process is *TU_AcceptS* shown in Figure 4.19 which creates the 200 OK to SUBSCRIBE message notifying the backup server that the subscription request has been accepted.

This block also contains two procedures which represent those messages that are sent repeatedly within the session. The first is the Notification Procedure called *Notification_Proc* shown in Figure 4.20 which issues a Notification message to the backup server. This is required after each change in state within the SIP Session and is therefore implemented as a procedure which can be repetitively instantiated. The procedure begins, creates the notification message, outputs the message to the backup server and then returns to start for the next change in state.



```
SIGNAL
Notify(charstring,charstring,charstring,charstring,charstring,charstring,charstring,charstring,charstring,charstring, charstring, charstring),
Subscribe(charstring,charstring,charstring,charstring,charstring,charstring,charstring,charstring,charstring,charstring,charstring,charstring),
AcceptRespN(charstring,charstring,charstring,charstring,charstring,charstring,charstring),
AcceptRespS(charstring,charstring,charstring,charstring,charstring,charstring,charstring), AliveP1,AliveP2,AliveB1,AliveB2,
RequestP(charstring,charstring,charstring,charstring,charstring,charstring,integer,charstring,charstring,charstring),
RequestB(charstring,charstring,charstring,charstring,charstring,charstring,integer,charstring,charstring,charstring),
ResponseB(charstring,charstring,charstring,charstring,charstring,charstring,charstring),
ResponseP(charstring,charstring,charstring,charstring,charstring,charstring,charstring),
Trying(charstring,charstring,charstring,charstring,charstring,charstring),
StateReq(charstring,charstring,charstring,charstring,charstring,charstring,charstring),
StateResp(charstring,charstring,charstring,charstring,charstring,charstring,charstring);
```

Figure 4.13 Block Model of Fault Tolerance Block

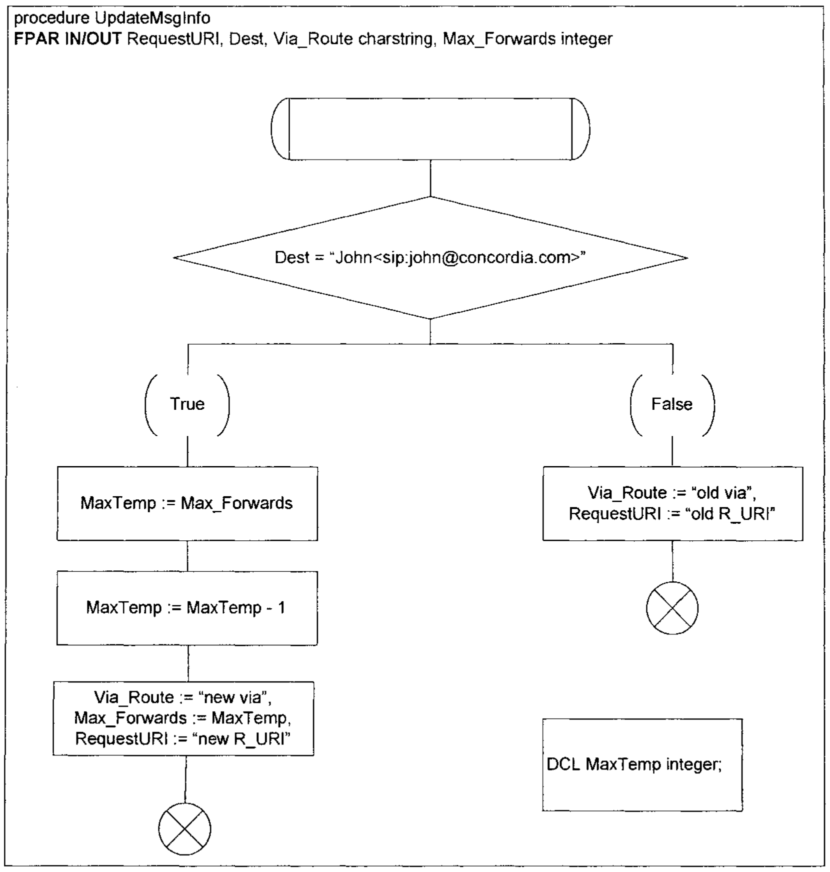

procedure UpdateMsgInfo
FPAR IN/OUT RequestURI, Dest, Via_Route charstring, Max_Forwards integer
Dest = "John<sip:john@concordia.com>"
True
False
MaxTemp := Max_Forwards
MaxTemp := MaxTemp - 1
Via_Route := "new via", Max_Forwards := MaxTemp, RequestURI := "new R_URI"
Via_Route := "old via", RequestURI := "old R_URI"
DCL MaxTemp integer;

Figure 4.14 Procedure to update header values in proxy server

The second procedure is the *StateReq_Proc* which creates the request that the primary issues to the backup server as a means of determining the current state of the session. This is executed every time the primary proxy server recovers from a failure within the SIP Session. This can occur several times within the system and so this consistency requirement is implemented as a procedure.

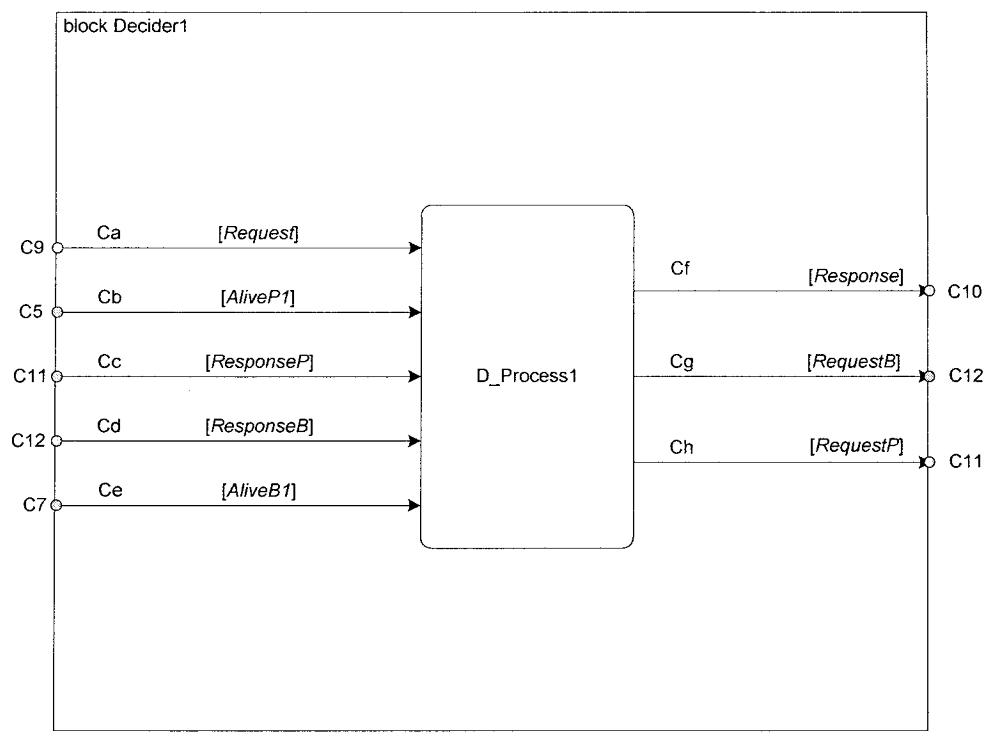


Figure 4.15 Block model of Decider Block attached to Calling Party or next proxy

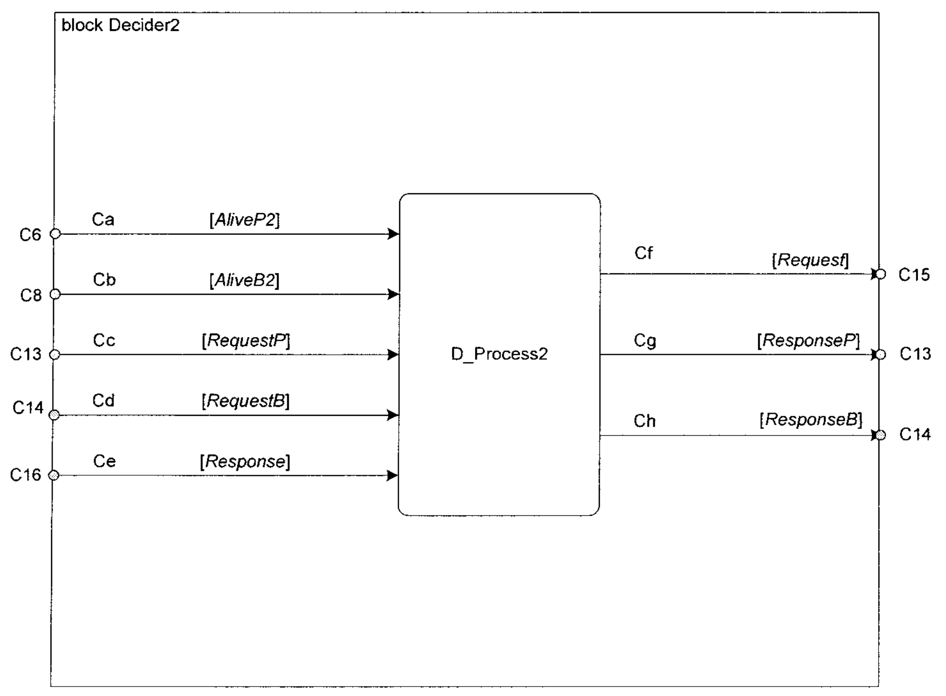


Figure 4.16 Block model of Decider Block attached to Called Party or next proxy

This block also contains two procedures which represent those messages that are sent repeatedly within the session. The first is the Notification Procedure called *Notification_Proc* shown in Figure 4.20 which issues a Notification message to the backup server. This is required after each change in state within the SIP Session and is therefore implemented as a procedure which can be repetitively instantiated. The procedure begins, creates the notification message, outputs the message to the backup server and then returns to start for the next change in state. The second procedure is the *StateReq_Proc* which creates the request that the primary issues to the backup server as a means of determining the current state of the session. This is executed every time the primary proxy server recovers from a failure within the SIP Session. This can occur several times within the system and so this consistency requirement is implemented as a procedure.

The final sub block is the backup proxy block shown in Figure 4.22. This contains the *Backup_Process* process which defines the workings of the backup proxy server. The block contains the channels of messages that flow through the proxy. There are two processes within this block. The first is *TU_Trying* which is the same as that found in the primary proxy server block. In the event that the backup takes over the session from the beginning, it must send a 100 TRYING to the end user agent. The second process is *TU_Sub* shown in Figure 4.23 which creates the SUBSCRIBE message and sends it to the primary proxy server thereby requesting that it be notified during the SIP Session Establishment and Termination phases.

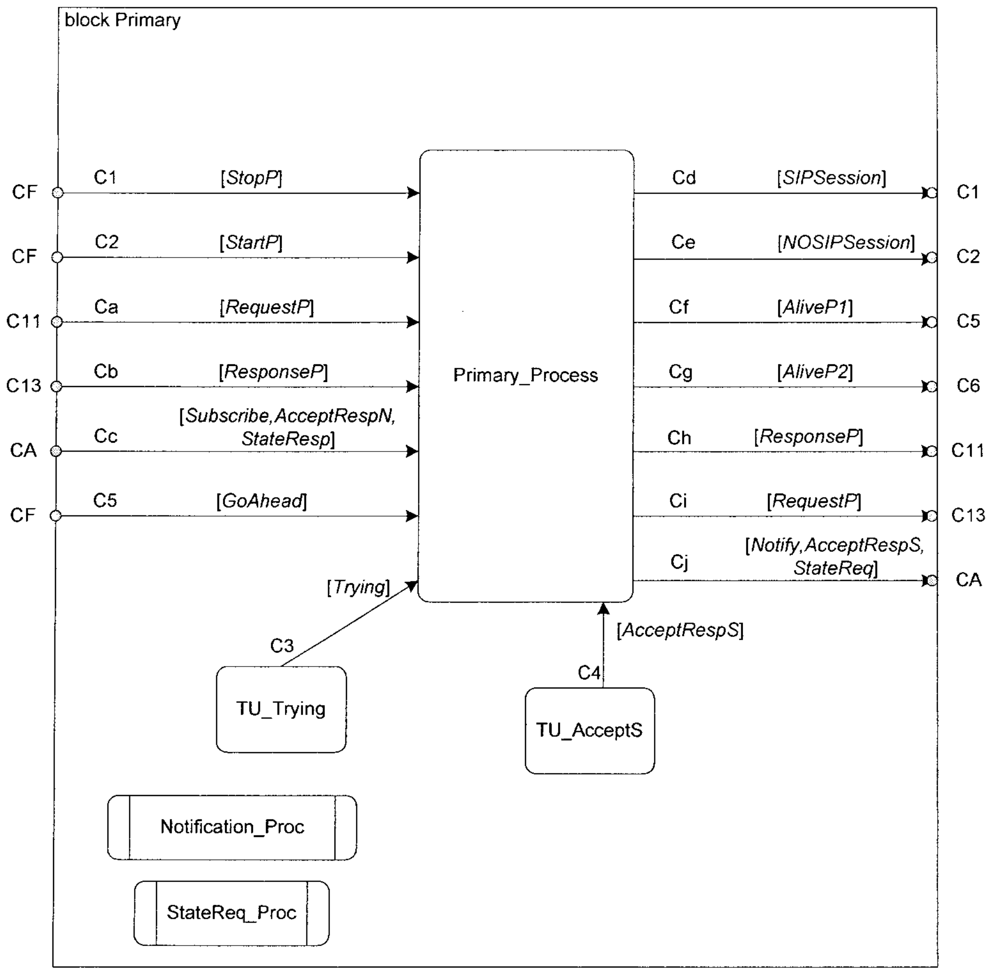


Figure 4.17 Block model of primary proxy server

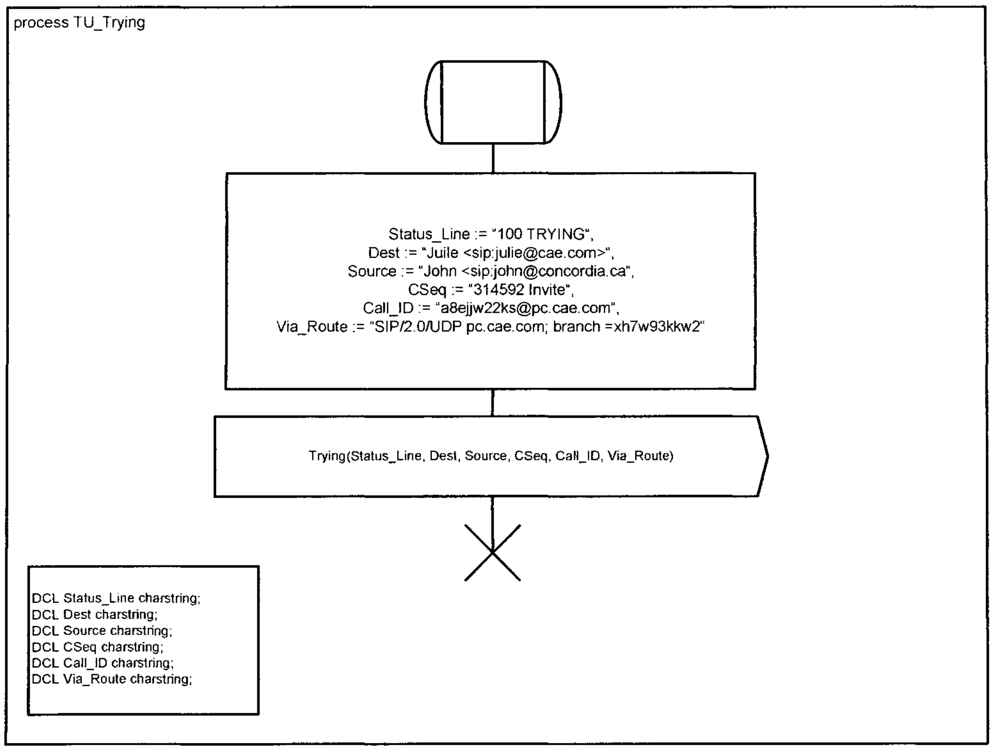

process TU_Trying
Status_Line := "100 TRYING",
Dest := "Juile <sip:julie@cae.com>",
Source := "John <sip:john@concordia.ca",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2"
Trying(Status_Line, Dest, Source, CSeq, Call_ID, Via_Route)
DCL Status_Line charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;

Figure 4.18 Creation of 100 TRYING Response

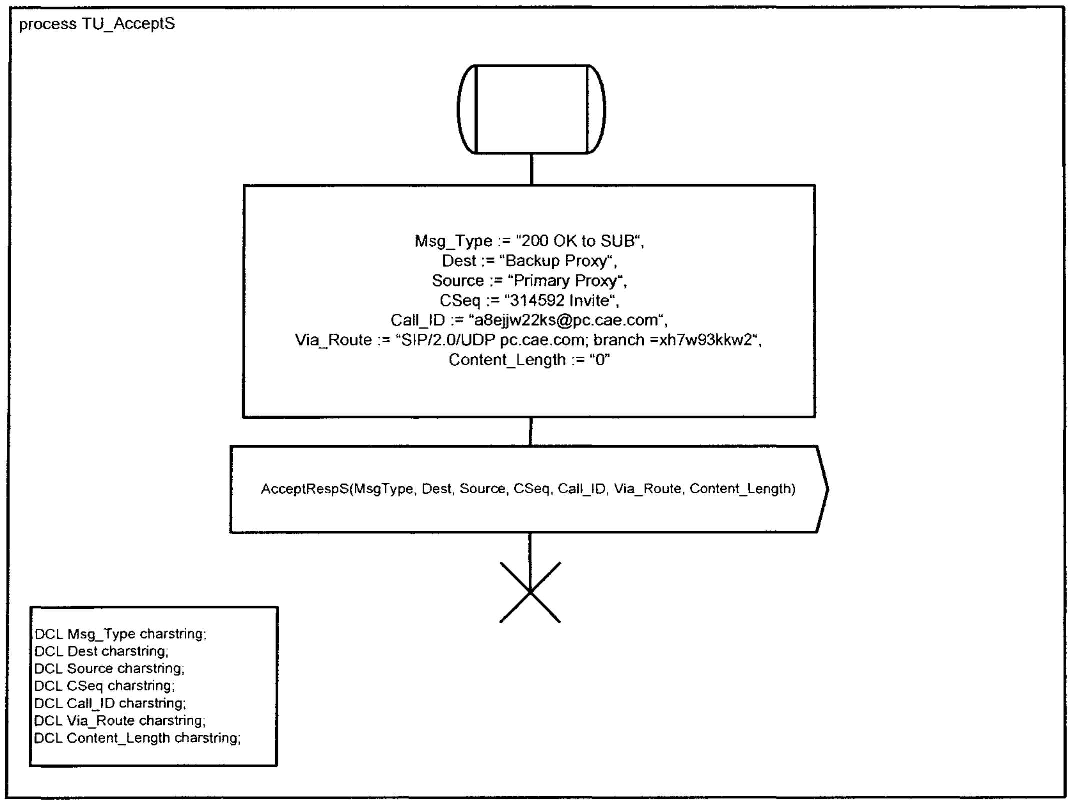

process TU_AcceptS
Msg_Type := "200 OK to SUB",
Dest := "Backup Proxy",
Source := "Primary Proxy",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2",
Content_Length := "0"
AcceptRespS(MsgType, Dest, Source, CSeq, Call_ID, Via_Route, Content_Length)
DCL Msg_Type charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;
DCL Content_Length charstring;

Figure 4.19 Creation of 200 OK to SUBSCRIBE Response

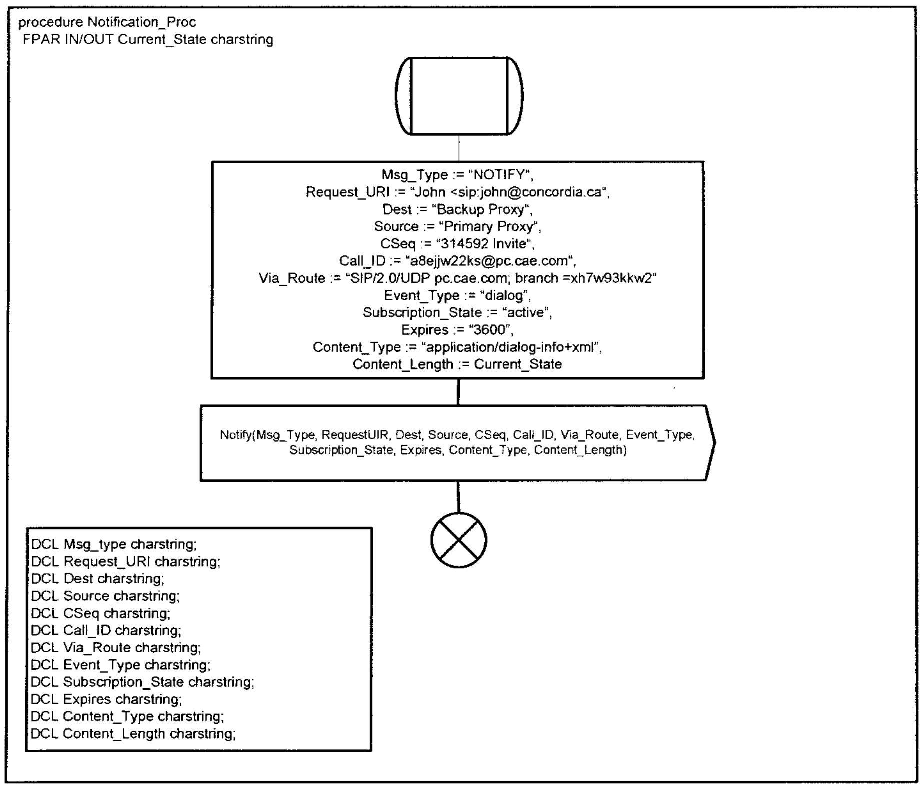

procedure Notification_Proc
FPAR IN/OUT Current_State charstring
Msg_Type := "NOTIFY",
Request_URI := "John <sip:john@concordia.ca",
Dest := "Backup Proxy",
Source := "Primary Proxy",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2"
Event_Type := "dialog",
Subscription_State := "active",
Expires := "3600",
Content_Type := "application/dialog-info+xml",
Content_Length := Current_State
Notify(Msg_Type, RequestUIR, Dest, Source, CSeq, Call_ID, Via_Route, Event_Type, Subscription_State, Expires, Content_Type, Content_Length)
DCL Msg_type charstring;
DCL Request_URI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;
DCL Event_Type charstring;
DCL Subscription_State charstring;
DCL Expires charstring;
DCL Content_Type charstring;
DCL Content_Length charstring;

Figure 4.20 Creation of NOTIFY Request

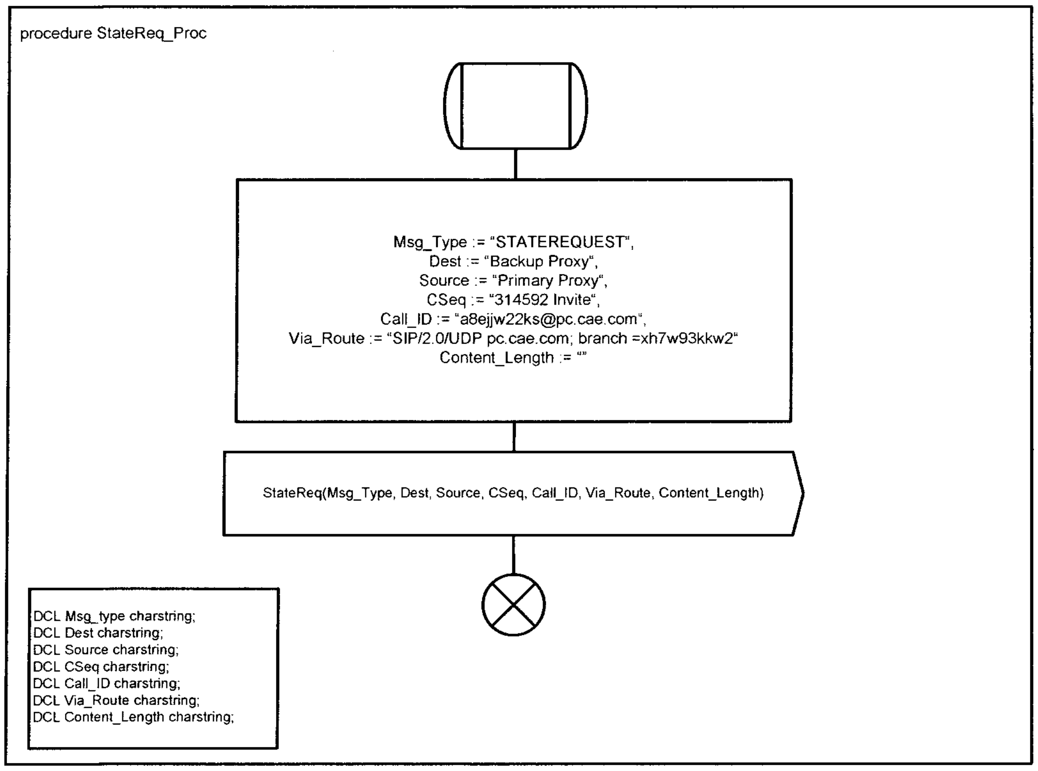

procedure StateReq_Proc
Msg_Type := "STATEREQUEST",
Dest := "Backup Proxy",
Source := "Primary Proxy",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2"
Content_Length := ""
StateReq(Msg_Type, Dest, Source, CSeq, Call_ID, Via_Route, Content_Length)
DCL Msg_type charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;
DCL Content_Length charstring;

Figure 4.21 Creation of StateRequest Request

This block also contains two procedures which represent the responses to the periodically received notification and state inquiry messages. The first is the *AcceptNotifiy_Proc* shown in Figure 4.24 which is the 200 OK to NOTIFY message issued to the primary proxy server. This is so that the primary proxy server can ensure that the backup received the change in state notification. Since the notification procedure occurs after every state, the backup proxy server must also accept the notification every time it receives the message and therefore it is also implemented as a structure which can be repeated at every call. The procedure begins, creates the acceptance to the notify message, outputs the message to the primary server and then returns to start for the next change in state. The second procedure is the *StateResp_Proc* shown in Figure 4.25 which creates the response to the request of the primary server. This message contains the current state of the SIP Session. The primary server can then read the value of the *Content_Length* header field found within the message to determine the current state. This is executed every time the primary proxy server recovers from a failure and sends a request for the current state of the session. This can occur several times within the system and is thus implemented as a procedure.

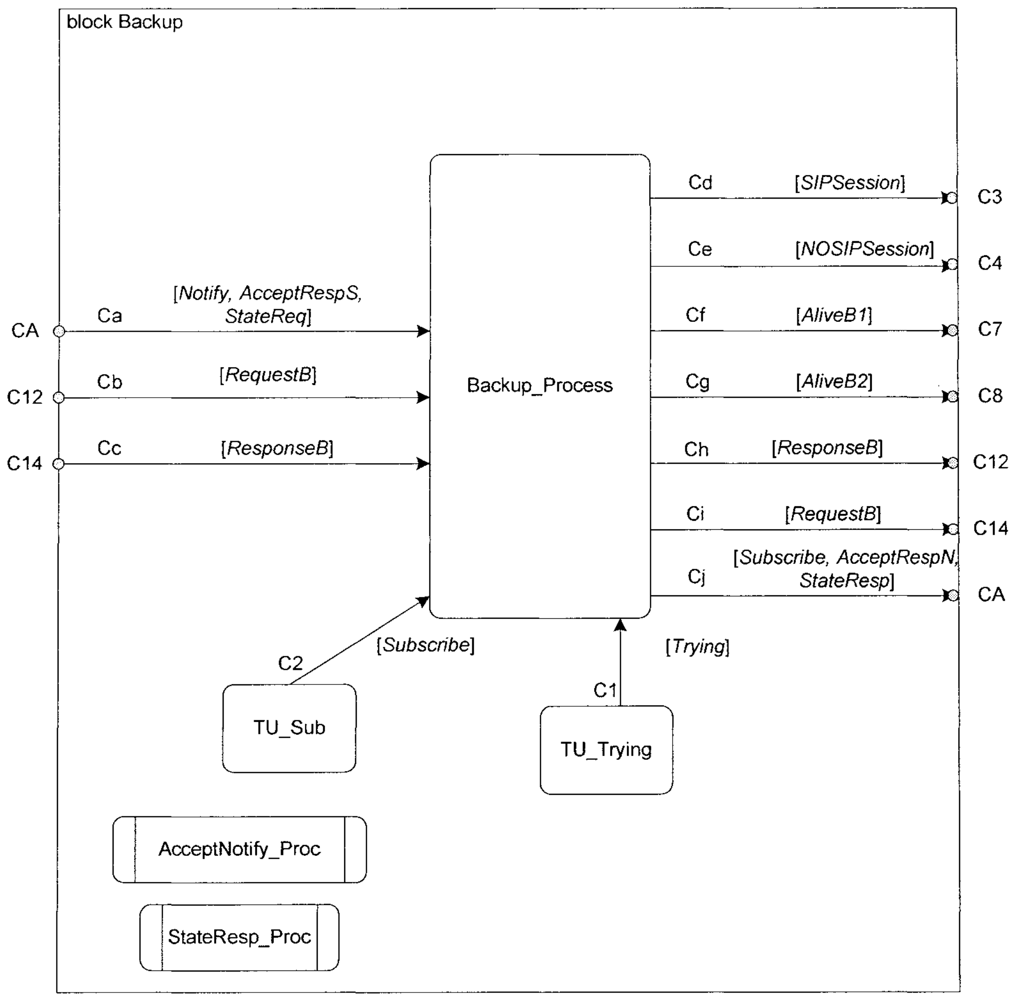
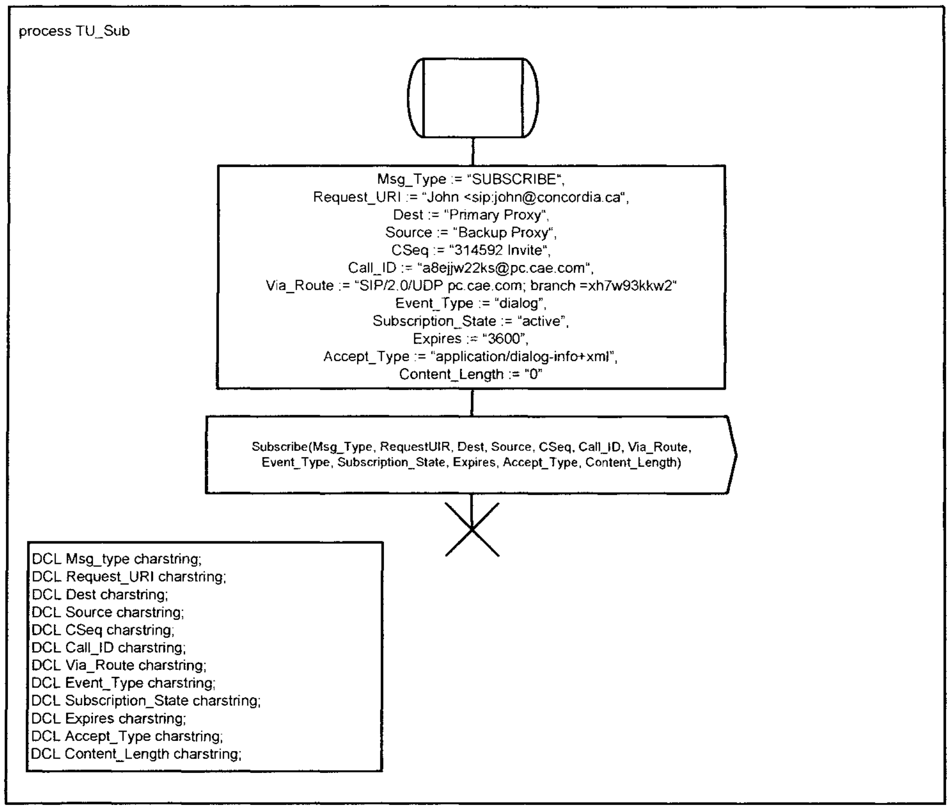


Figure 4.22 Block model of backup proxy server

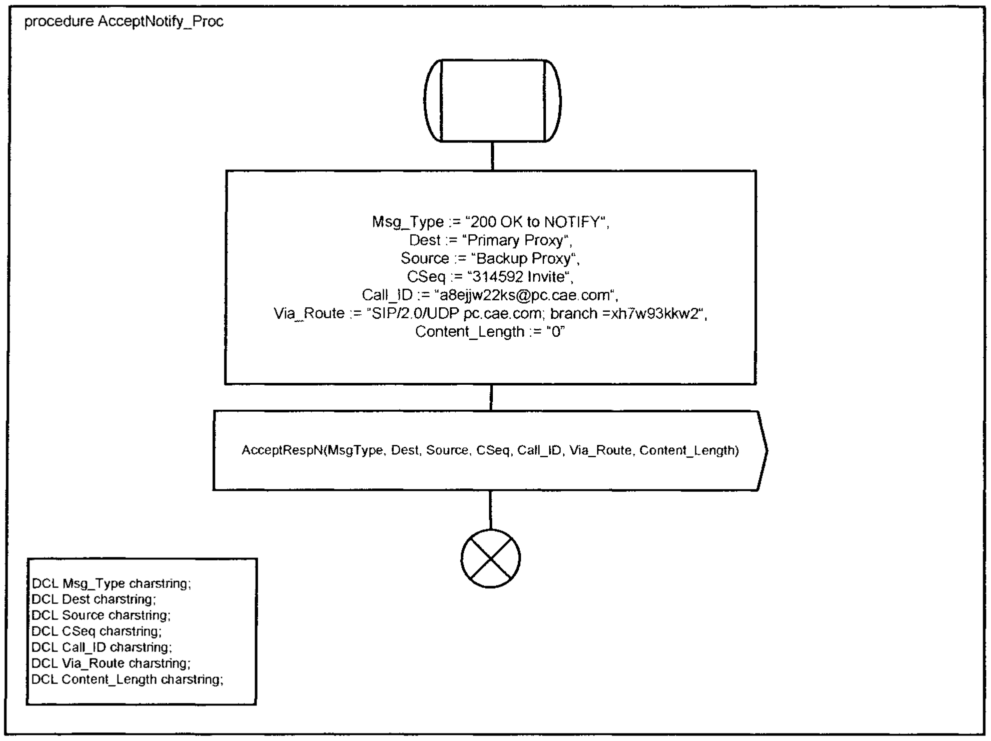


```
Msg_Type := "SUBSCRIBE",
Request_URI := "John <sip:john@concordia.ca",
Dest := "Primary Proxy",
Source := "Backup Proxy",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2"
Event_Type := "dialog",
Subscription_State := "active",
Expires := "3600",
Accept_Type := "application/dialog-info+xml",
Content_Length := "0"
```

```
Subscribe(Msg_Type, RequestUIR, Dest, Source, CSeq, Call_ID, Via_Route,
Event_Type, Subscription_State, Expires, Accept_Type, Content_Length)
```

```
DCL Msg_type charstring;
DCL Request_URI charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;
DCL Event_Type charstring;
DCL Subscription_State charstring;
DCL Expires charstring;
DCL Accept_Type charstring;
DCL Content_Length charstring;
```

Figure 4.23 Creation of SUBSCRIBE Request

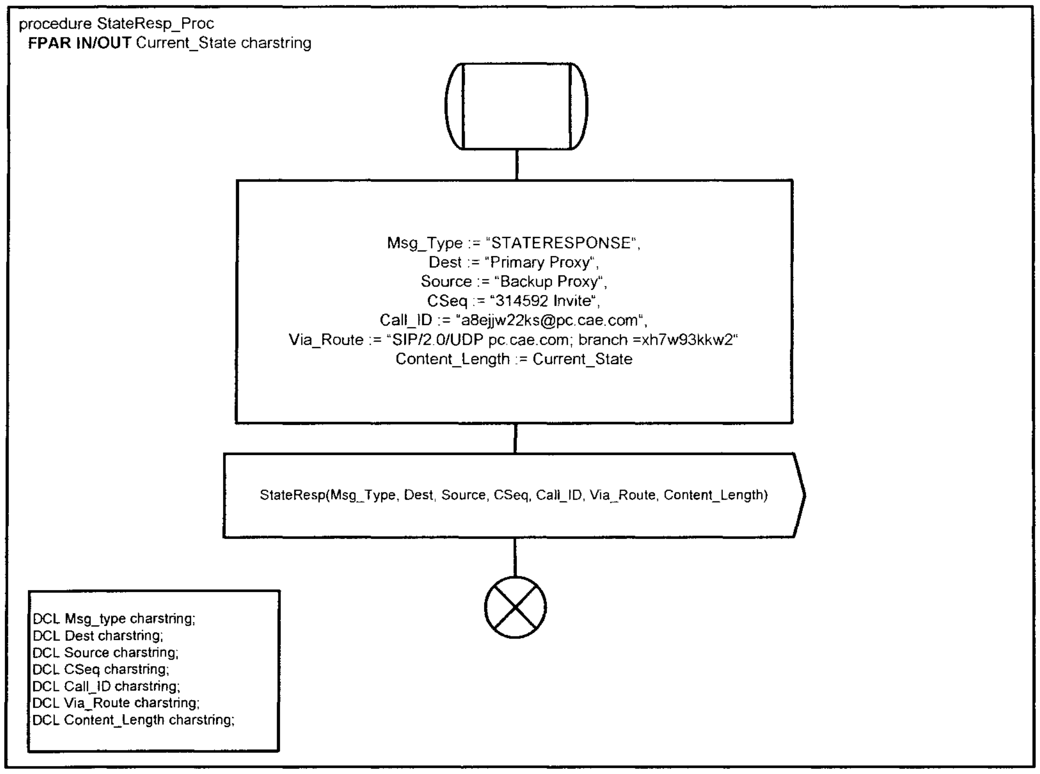


```
Msg_Type := "200 OK to NOTIFY",
Dest := "Primary Proxy",
Source := "Backup Proxy",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2",
Content_Length := "0"
```

```
AcceptRespN(MsgType, Dest, Source, CSeq, Call_ID, Via_Route, Content_Length)
```

```
DCL Msg_Type charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;
DCL Content_Length charstring;
```

Figure 4.24 Creation of 200 OK to NOTIFY Response



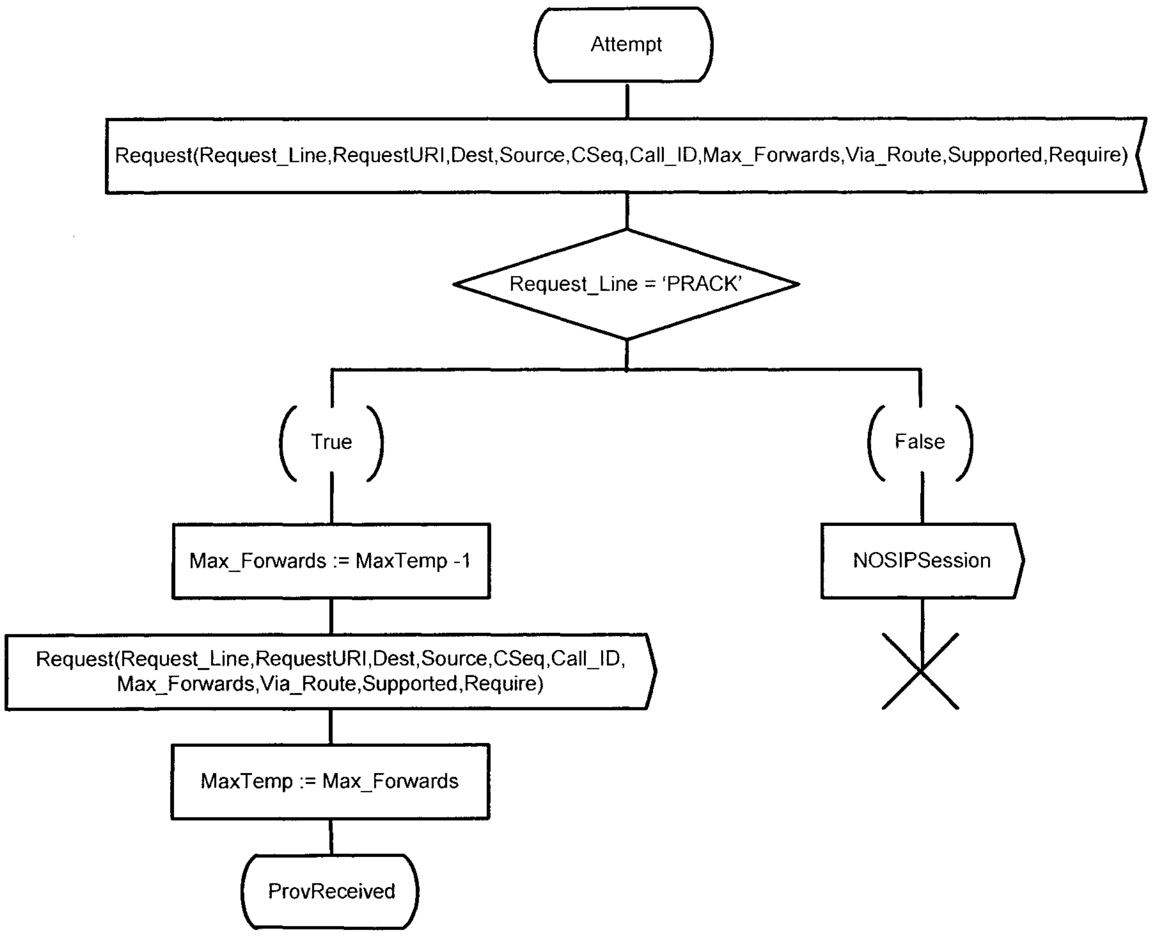
procedure StateResp_Proc
FPAR IN/OUT Current_State charstring
Msg_Type := "STATERESPONSE",
Dest := "Primary Proxy",
Source := "Backup Proxy",
CSeq := "314592 Invite",
Call_ID := "a8ejjw22ks@pc.cae.com",
Via_Route := "SIP/2.0/UDP pc.cae.com; branch =xh7w93kkw2"
Content_Length := Current_State
StateResp(Msg_Type, Dest, Source, CSeq, Call_ID, Via_Route, Content_Length)
DCL Msg_type charstring;
DCL Dest charstring;
DCL Source charstring;
DCL CSeq charstring;
DCL Call_ID charstring;
DCL Via_Route charstring;
DCL Content_Length charstring;

Figure 4.25 Creation of StateResponse Response

4.4 EFSM Level Process Design

Each process within SDL is defined through an EFSM. This contains the workings of the process as described theoretically in Section 3.4.2. Within each diagram, a change in state is dependent on the receipt of a message or the sending of a message to another process within the system.

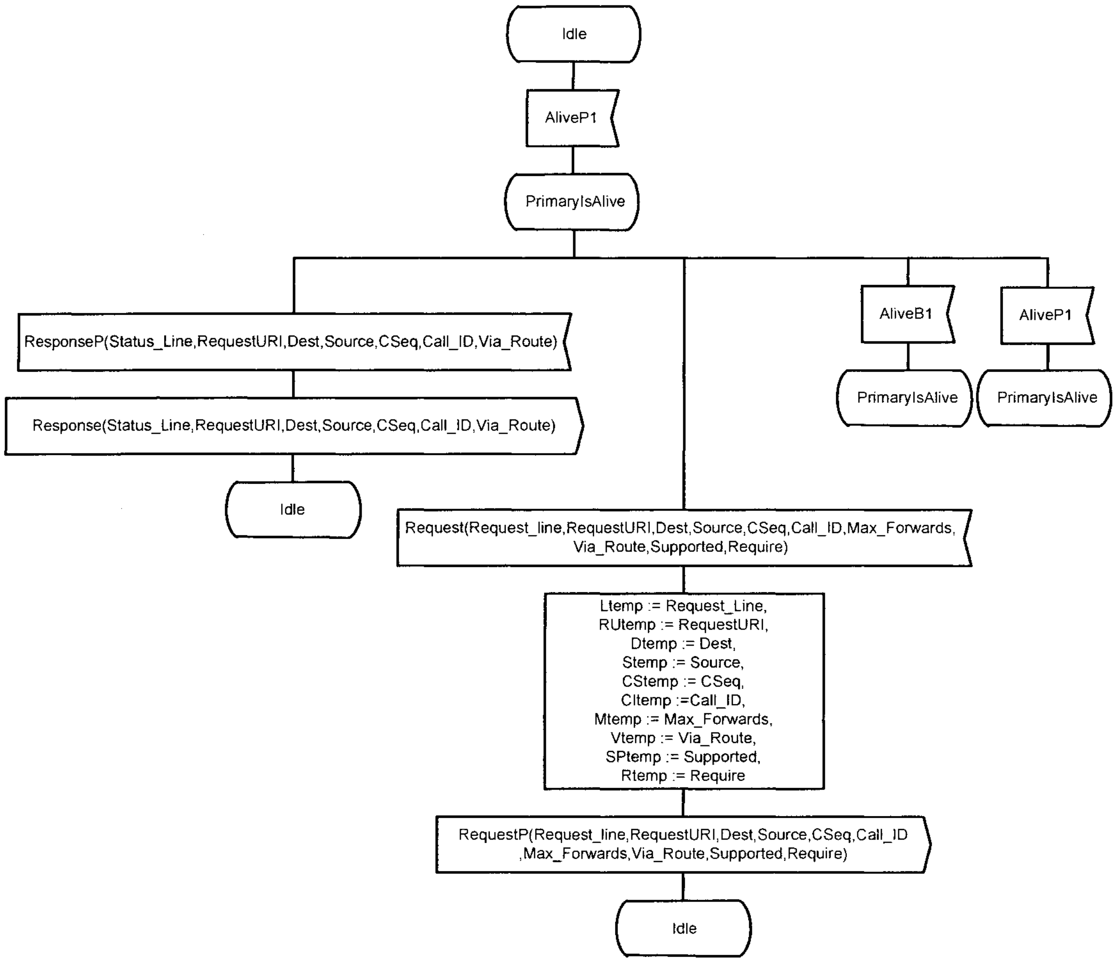
A part of the EFSM for the *SIP_Sender* process is shown in Figure 4.26. The scenario shown describes the Calling Party in the *Attempt* state. It has received the provisional response and is now awaiting the receipt of a PRACK message from its transaction user. Upon receipt, the *Max_Forwards* value is decreased and the message is sent to the Decider block. It then goes to the *ProvReceived* state. If the message received is not PRACK, there is no SIP session established and the EFSM terminates.



Attempt
Request(Request_Line,RequestURI,Dest,Source,CSeq,Call_ID,Max_Forwards,Via_Route,Supported,Require)
Request_Line = 'PRACK'
True
False
Max_Forwards := MaxTemp -1
NOSIPSession
Request(Request_Line,RequestURI,Dest,Source,CSeq,Call_ID, Max_Forwards,Via_Route,Supported,Require)
MaxTemp := Max_Forwards
ProvReceived

Figure 4.26 Partial EFSM of *SIP_Sender* process

A part of the EFSM for the *D_Process1* process is shown in Figure 4.27. The scenario shown describes the Decider process receiving a Request from the Calling Party and then issuing it to the primary proxy server. Note that before sending the message, the process saves the Request in case it has to be retransmitted to the backup proxy server. The other branch shows the Decider process receiving a Response from the primary proxy and then sending it to the Calling Party. If any Alive messages are received while in the *PrimaryIsAlive* state, the messages do not alter the state of the Decider. This ensures that the proper state is taken regardless of the order in which the *Alive* messages are received from the primary and the backup proxy servers.



Idle
AliveP1
PrimaryIsAlive
ResponseP(Status_Line,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route)
Response(Status_Line,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route)
Idle
AliveB1
PrimaryIsAlive
AliveP1
PrimaryIsAlive
Request(Request_line,RequestURI,Dest,Source,CSeq,Call_ID,Max_Forwards, Via_Route,Supported,Require)
Ltemp := Request_Line,
RUtemp := RequestURI,
Dtemp := Dest,
Stemp := Source,
CStemp := CSeq,
CItemp :=Call_ID,
Mtemp := Max_Forwards,
Vtemp := Via_Route,
SPtemp := Supported,
Rtemp := Require
RequestP(Request_line,RequestURI,Dest,Source,CSeq,Call_ID ,Max_Forwards,Via_Route,Supported,Require)
Idle

Figure 4.27 Partial EFSM of *D_Process1* process

Similarly, Figure 4.28 shows a part of the *D_Process2* process which describes the Decider process receiving the messages from the backup proxy server and then sending it to the respective parties. The scenario shown is the receipt of a request from the Calling party. The Decider block waits for a predetermined time for the *AliveP* message from the primary proxy server. If it is not received, the Decider process issues the Request to the backup proxy server. Since the receipt of messages by the Decider server is not controlled, the *Alive* messages may arrive before the timeout occurs. These situations are handled within the EFSM of the Decider process.

The branch which takes a timer Tk value as an input is required in order to retransmit lost messages. This occurs if the primary proxy server fails after the Decider has issued a message to it. In this case, no response will be received and the message times out. At this point, a fault is detected within the primary proxy server and the message will then be retransmitted to the backup proxy server which takes over the current session.

Before this retransmission occurs however the Calling or Called parties can send the next message since their processes are independent of the Fault tolerance block. The provision taken is to input the message, store it into a buffer and set a flag indicating that the next message has been stored. When the next alive message is received, it indicates that the previous message has been retransmitted successfully. At this point, the buffer is checked. If it is empty, the Decider waits for a UA to send a message. In the case where the buffer is full, the saved message is transmitted to the appropriate proxy server.

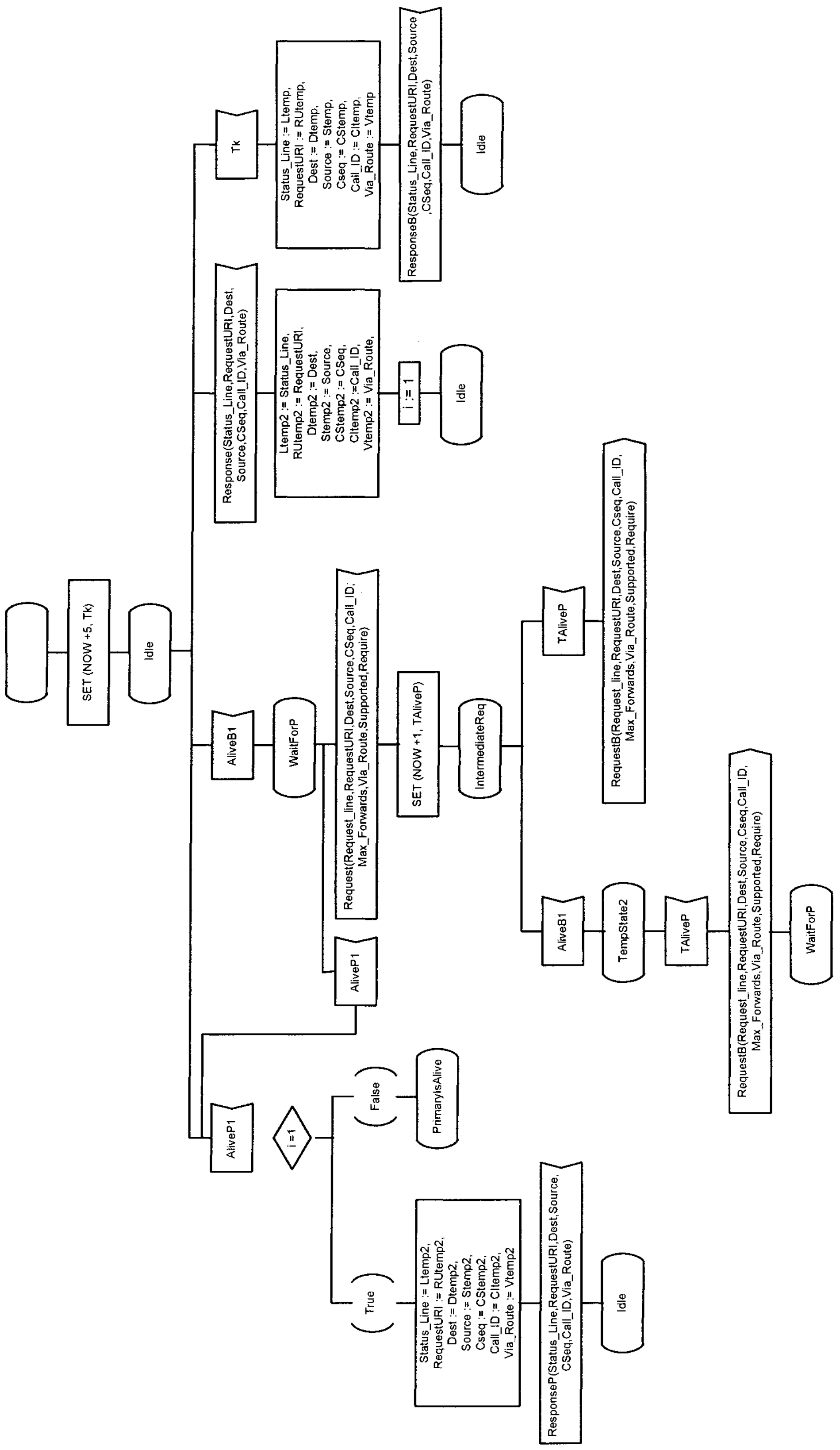
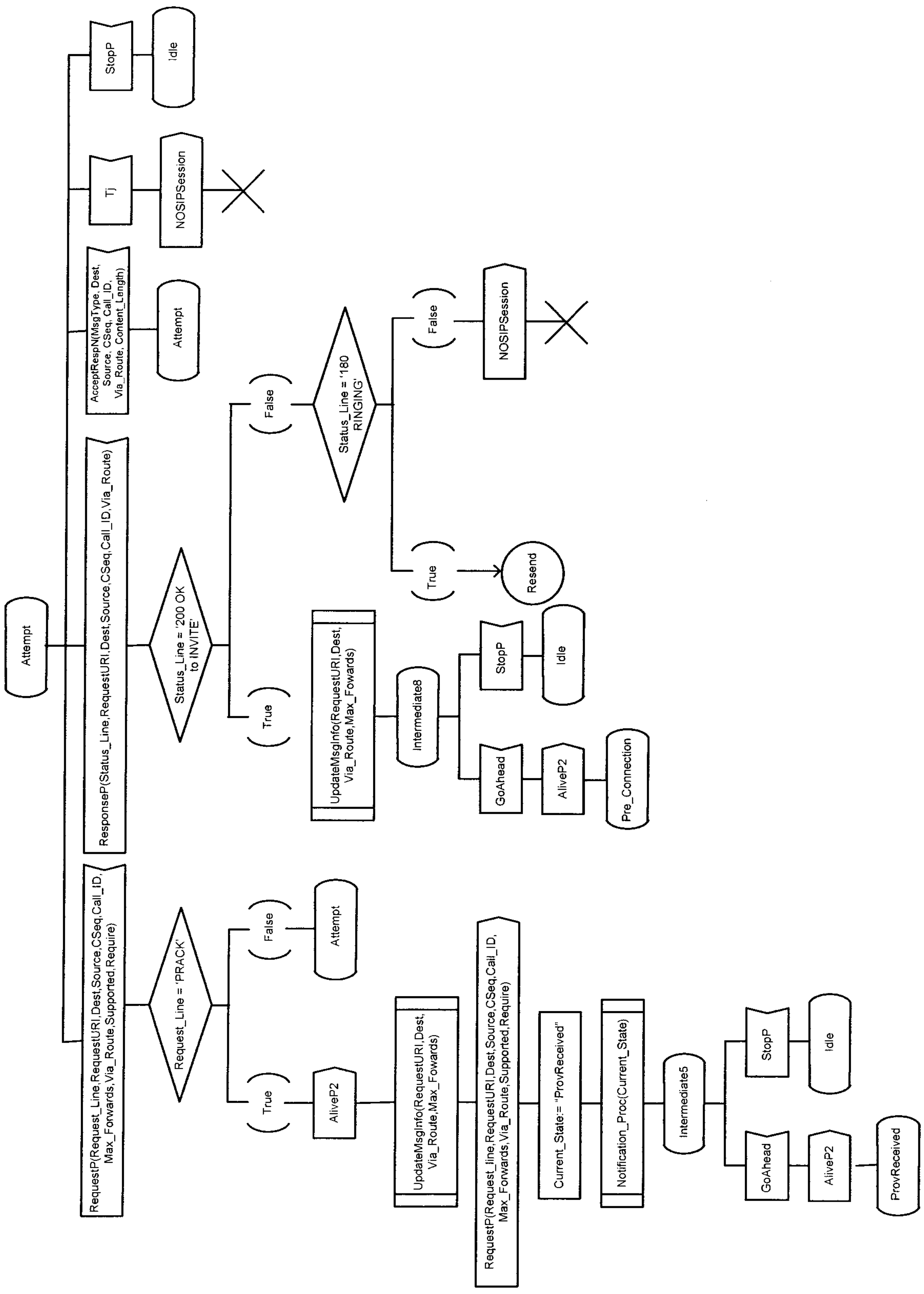

SET (NOW +5, Tk)
Idle
Response(Status_Line,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route)
Ltemp2 := Status_Line, RUtemp2 := RequestURI, Dtemp2 := Dest, Stemp2 := Source, CStemp2 := CSeq, CItemp2 :=Call_ID, Vtemp2 := Via_Route,
i := 1
Idle
Tk
Status_Line := Ltemp, RequestURI := RUtemp, Dest := Dtemp, Source := Stemp, Cseq := CStemp, Call_ID := CItemp, Via_Route := Vtemp
ResponseB(Status_Line,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route)
Idle
AliveB1
WaitForP
Request(Request_line,RequestURI,Dest,Source,CSeq,Call_ID,Max_Forwards,Via_Route,Supported,Require)
SET (NOW +1, TAliveP)
IntermediateReq
TAliveP
RequestB(Request_line,RequestURI,Dest,Source,Cseq,Call_ID,Max_Forwards,Via_Route,Supported,Require)
AliveB1
TempState2
TAliveP
RequestB(Request_line,RequestURI,Dest,Source,Cseq,Call_ID,Max_Forwards,Via_Route,Supported,Require)
WaitForP
AliveP1
AliveP1
i =1
False
PrimaryIsAlive
True
Status_Line := Ltemp2, RequestURI := RUtemp2, Dest := Dtemp2, Source := Stemp2, Cseq := CStemp2, Call_ID := CItemp2, Via_Route := Vtemp2
ResponseP(Status_Line,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route)
Idle

Figure 4.28 Partial EFSM of *D_Process2* process

A part of the EFSM for the *Primary_Process* process is shown in Figure 4.29. The primary proxy server is in the *Attempt* state. At this stage, a PRACK message can be received if the extension is supported within the session. This will cause the EFSM to go to the *ProvReceived* state. Prior to the transfer to the new state, the primary proxy server may fail. This is simulated using the *StopP* message inputted from the environment. In order to simulate proper functionality of the system, the *GoAhead* message is inputted. If no PRACK is used, then the 200 OK to INVITE Acceptance Response may be received directly. At this point the decision to simulate a fault or not is again possible. If neither of the responses are received then the 180 Ringing message times out and no SIP session is established. The EFSM terminates in this case.

Within this EFSM, it is assumed that the backup proxy server has subscribed to the primary and therefore notification messages are sent when the EFSM changes state. The Acceptance Response to the notification can be received *AcceptRespS* however the state of the server is not altered. This subscription however can be done at any point within the session. The EFSM for the case wherein the subscription is not done at the beginning is slightly different. In the branch where PRACK is received, no notification message is sent and instead of accepting the response to the notification, the subscription message is accepted as shown in Figure 4.30. The states are appended with a '2' as *Attempt2*. After the subscription is accepted, the EFSM reverts to the Attempt state wherein notification takes place. The primary proxy continues to notify the backup for the rest of the session.



Attempt
StopP
Idle
Tj
NOSIPSession
AcceptRespN(MsgType, Dest, Source, CSeq, Call_ID, Via_Route, Content_Length)
Attempt
ResponseP(Status_Line,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route)
Status_Line = '200 OK to INVITE'
False
Status_Line = '180 RINGING'
False
NOSIPSession
True
Resend
True
UpdateMsgInfo(RequestURI,Dest, Via_Route,Max_Fowards)
Intermediate8
StopP
Idle
GoAhead
AliveP2
Pre_Connection
RequestP(Request_Line,RequestURI,Dest,Source,CSeq,Call_ID, Max_Forwards,Via_Route,Supported,Require)
Request_Line = 'PRACK'
False
Attempt
True
AliveP2
UpdateMsgInfo(RequestURI,Dest, Via_Route,Max_Fowards)
RequestP(Request_line,RequestURI,Dest,Source,CSeq,Call_ID, Max_Forwards,Via_Route,Supported,Require)
Current_State:= "ProvReceived"
Notification_Proc(Current_State)
Intermediate5
StopP
Idle
GoAhead
AliveP2
ProvReceived

Figure 4.29 Partial EFSM of *Primary_Process* process

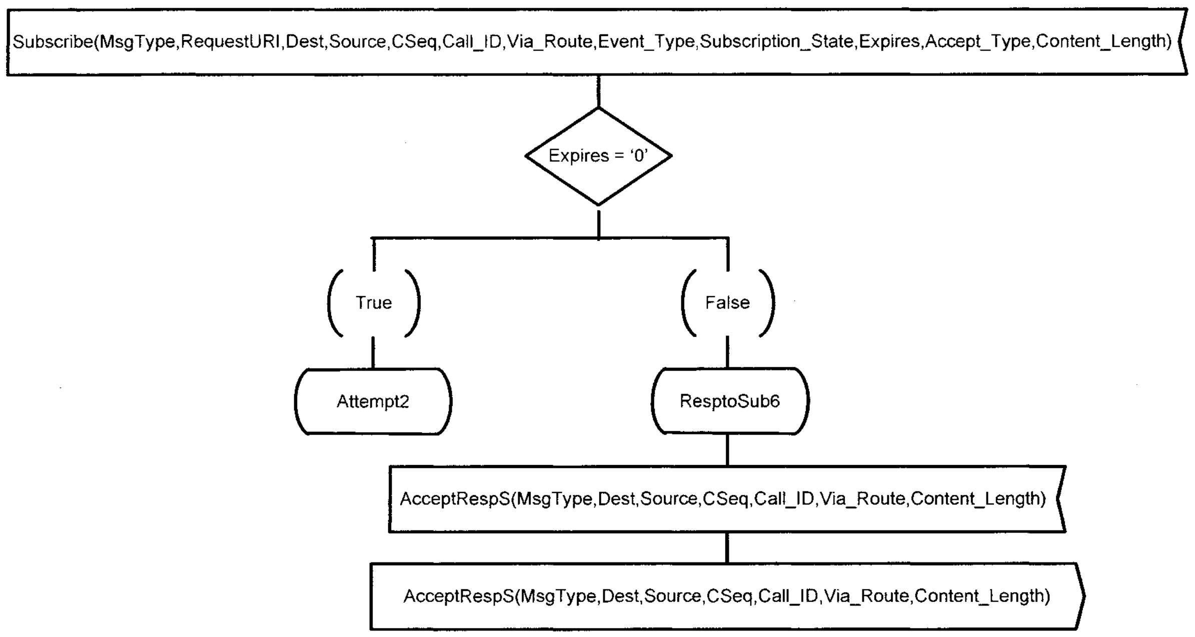

Subscribe(MsgType,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route,Event_Type,Subscription_State,Expires,Accept_Type,Content_Length)
Expires = '0'
True
False
Attempt2
ResptoSub6
AcceptRespS(MsgType,Dest,Source,CSeq,Call_ID,Via_Route,Content_Length)
AcceptRespS(MsgType,Dest,Source,CSeq,Call_ID,Via_Route,Content_Length)

Figure 4.30 Additional transition for EFSM of *Primary_Process* process

A part of the EFSM for the *Backup_Process* process is shown in Figure 4.31. The scenario shown here is the same as that in the primary proxy server however the backup can either receive the messages directly or else it can transfer to the correct state with the notification message. The state sent by the primary proxy server is checked and the backup proxy server goes to the given state. This assures consistency between both servers.

At each state within this EFSM, the backup may receive a Request from the primary proxy server as to the current state of the session. This occurs after the primary server recovers from a fault. The Response is sent by the backup server informing the primary of the current state of the session although the state of the backup server is not changed. This change will only occur once a notification message is received from the primary or else a Request or Response from the Decider block.

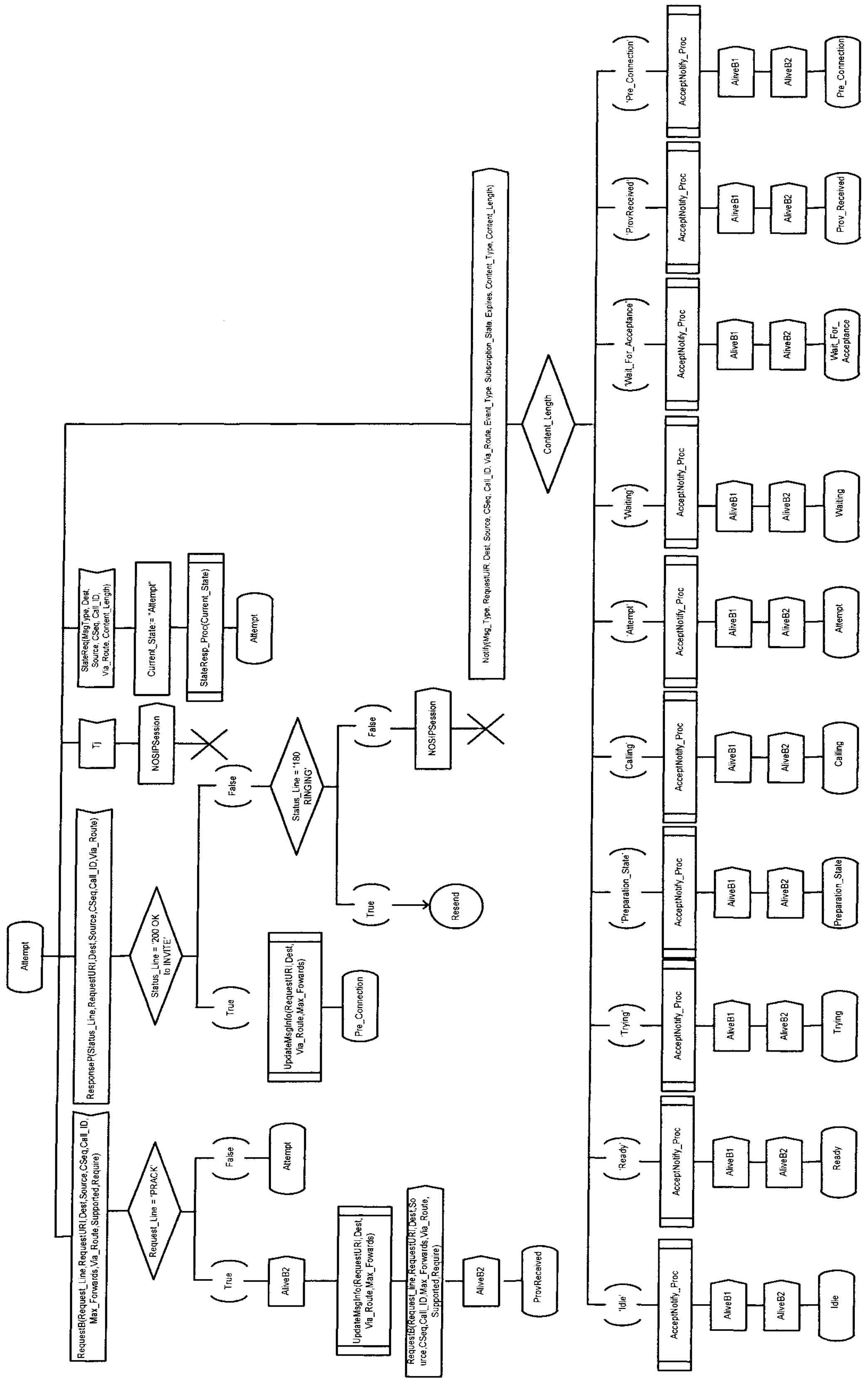

Attempt
RequestB(Request_Line,RequestURI,Dest,Source,CSeq,Call_ID,Max_Forwards,Via_Route,Supported,Require)
Request_Line = 'PRACK'
True
False
AliveB2
Attempt
UpdateMsgInfo(RequestURI,Dest,Via_Route,Max_Fowards)
RequestB(Request_line,RequestURI,Dest,Source,CSeq,Call_ID,Max_Forwards,Via_Route,Supported,Require)
ProvReceived
ResponseP(Status_Line,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route)
Status_Line = '200 OK to INVITE'
Pre_Connection
Status_Line = '180 RINGING'
Resend
NOSIPSession
TI
StateReq(MsgType,Dest,Source,CSeq,Call_ID,Via_Route,Content_Length)
Current_State := "Attempt"
StateResp_Proc(Current_State)
Notify(Msg_Type,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route,Event_Type,Subscription_State,Expires,Content_Type,Content_Length)
Content_Length
'Idle'
'Ready'
'Trying'
'Preparation_State'
'Calling'
'Attempt'
'Waiting'
'Wait_For_Acceptance'
'ProvReceived'
'Pre_Connection'
AcceptNotify_Proc
AliveB1
AliveB2
Idle
Ready
Trying
Preparation_State
Calling
Waiting
Wait_For_Acceptance
Prov_Received

Figure 4.31 Partial EFSM of *Backup_Process* process

If the backup proxy server is not subscribed to the primary proxy server, then the notification branch is replaced with the one in Figure 4.32. The Subscribe message is taken as an input from its own process which creates the message (*TU_Sub*). The message is then sent to the primary proxy server.

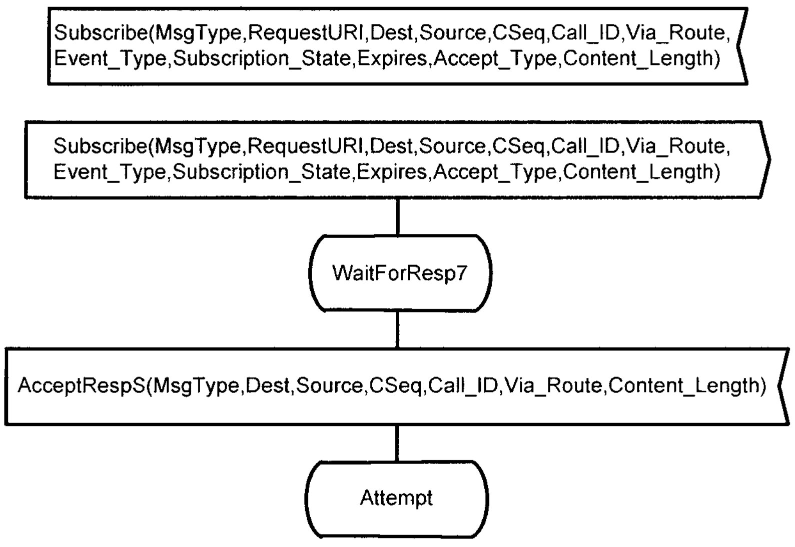

Subscribe(MsgType,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route, Event_Type,Subscription_State,Expires,Accept_Type,Content_Length)
Subscribe(MsgType,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route, Event_Type,Subscription_State,Expires,Accept_Type,Content_Length)
WaitForResp7
AcceptRespS(MsgType,Dest,Source,CSeq,Call_ID,Via_Route,Content_Length)
Attempt

Figure 4.32 Additional transition for EFSM of *Backup_Process* process

A part of the EFSM for the *SIP_Receiver* process is shown in Figure 4.33. The scenario shown describes the Called Party in the *Pre_Connection* state. It has sent the acceptance response and is now awaiting the receipt of an ACK message from the Calling Party. Upon receipt, the Called Party sends out a *SIPSession* message to the environment stating that the session has been correctly established. The EFSM then returns to the initial *Idle* state awaiting a new session invitation. In the case where the message is a BYE Request instead of an ACK, the state machine jumps to the *Bye_Check* point wherein the Request is consumed and the state of the machine changes accordingly.

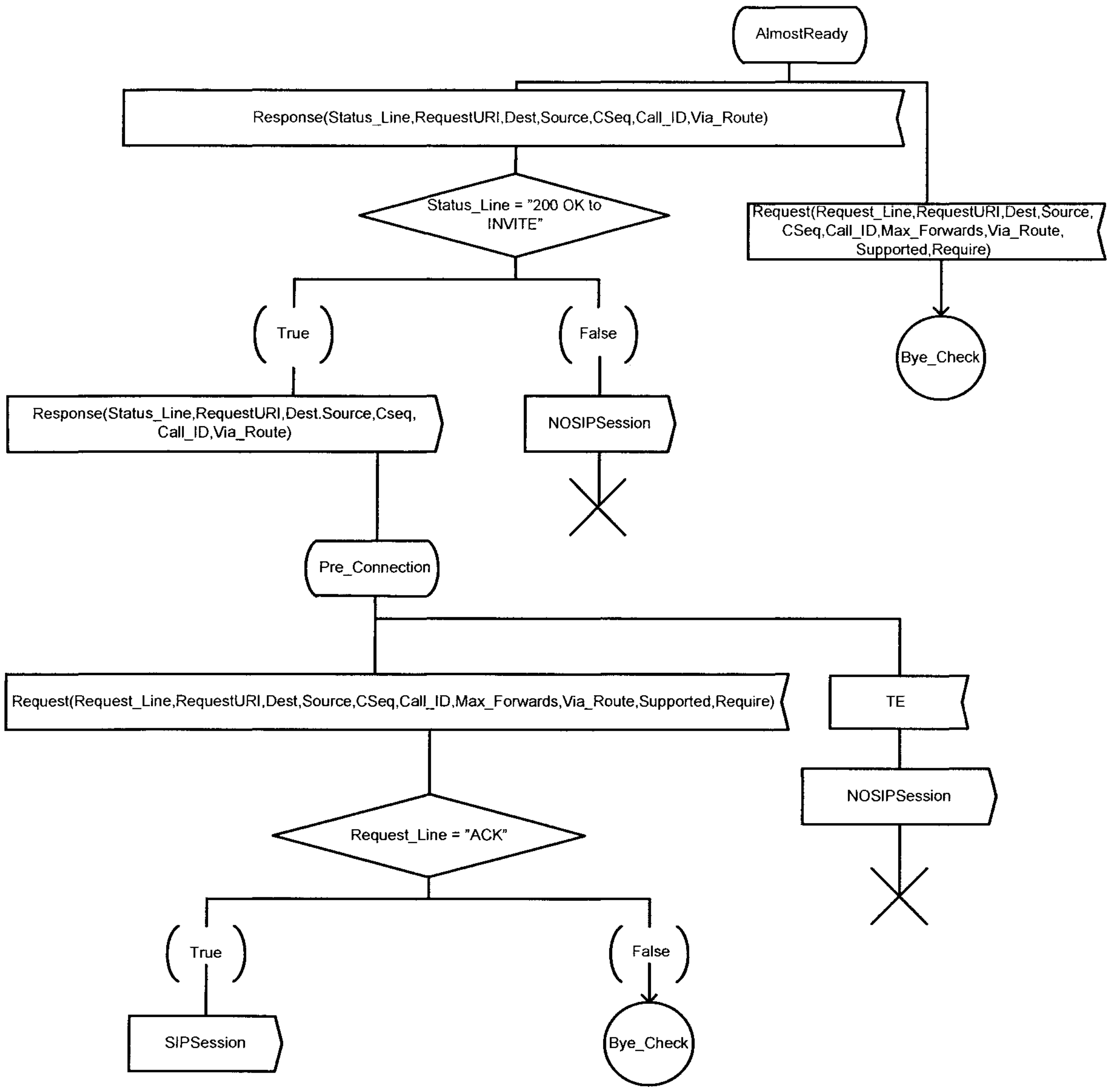

AlmostReady
Response(Status_Line,RequestURI,Dest,Source,CSeq,Call_ID,Via_Route)
Status_Line = "200 OK to INVITE"
Request(Request_Line,RequestURI,Dest,Source, CSeq,Call_ID,Max_Forwards,Via_Route, Supported,Require)
True
False
Bye_Check
Response(Status_Line,RequestURI,Dest.Source,Cseq, Call_ID,Via_Route)
NOSIPSession
Pre_Connection
Request(Request_Line,RequestURI,Dest,Source,CSeq,Call_ID,Max_Forwards,Via_Route,Supported,Require)
TE
NOSIPSession
Request_Line = "ACK"
True
False
SIPSession
Bye_Check

Figure 4.33 Partial EFSM of *SIP_Receiver* process

4.5 Timing Analysis

As mentioned in Chapter 3, the SIP protocol contains a set of timers applied to the requests and responses sent within the session. These timers are used to limit the waiting time of the end users. If a response is not received within the allotted time, the request expires and is either retransmitted or no SIP session is established. Recall that SIP messages can be sent along either TCP or UDP. The timeout values for both transportation protocols as defined in [4] are given for the end user agents in Table 4.2 and for the proxy servers in Table 4.3. Note that the maximum wait time for requests other than INVITE and responses to the INVITE is 4 sec. However with the usage of PRACK, this condition is uplifted so as to allow the retransmission of Provisional responses. Therefore, this timing constraint is not placed within our implementation as we enforce the usage of PRACK.

Furthermore, our implementation can be executed with the transport being UDP or TCP. In the case of UDP, timers can be expired thereby simulating the retransmission of the messages within a SIP session. This real-time feature is provided in SDL. In the case of TCP, no retransmission is required since this is a reliable transport protocol.

The final timing constraint is that placed on the retransmission of responses and is set to 500 milliseconds (Tk). In [4] this value is given to the proxy server as the wait time before a response should be retransmitted. In our implementation, this retransmission is executed by the Decider server and thus the timer expires within the Decider block for both the requests and responses sent between end user agents. A diagrammatic view of these timing constraints is shown in Figures 4.34 and 4.35.

	UDP Timeout	TCP Timeout
Timeout for INVITE Request [Tb]	32 sec	32 sec
Wait time for Response Retransmit [Td]	1 sec	No retransmission needed (0 sec)
Wait time for Request Retransmit [Te]	500 ms	No retransmission needed (0 sec)
Timeout for Responses (if PRACK used) [Tf]	32 sec	32 sec

Table 4.2 Timing Constraints for end users

	UDP Timeout	TCP Timeout
Timeout for ACK Request [Th]	32 sec	No retransmission needed (0 sec)
Wait time for ACK Retransmits [Ti]	5 sec	No retransmission needed (0 sec)
Wait time for Request Retransmit [Tj]	32 sec	No retransmission needed (0 sec)
Wait time for Response Retransmit [Tk]	500 msec	No retransmission needed (0 sec)

Table 4.3 Timing Constraints for proxy servers

The recovery process within our proposed approach is achieved through less than the 32 sec of the initial INVITE message. Faults are detected dynamically and thus the system need not restart from the beginning. The 32 sec timeout is reset by the calling party when the Provisional or Error response is received.

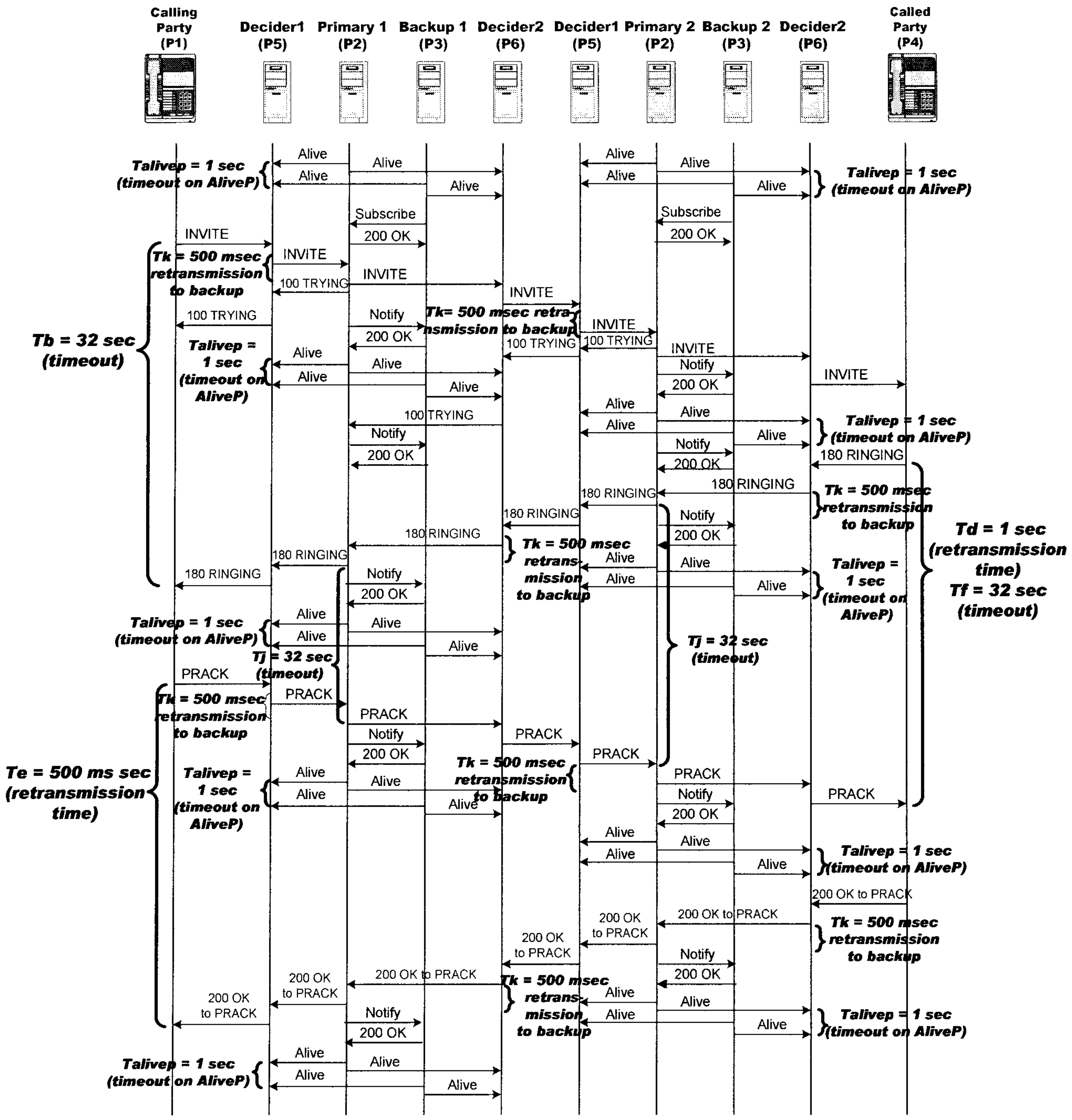


Figure 4.34 SIP Message Exchange with Timing Constraints (Part 1)

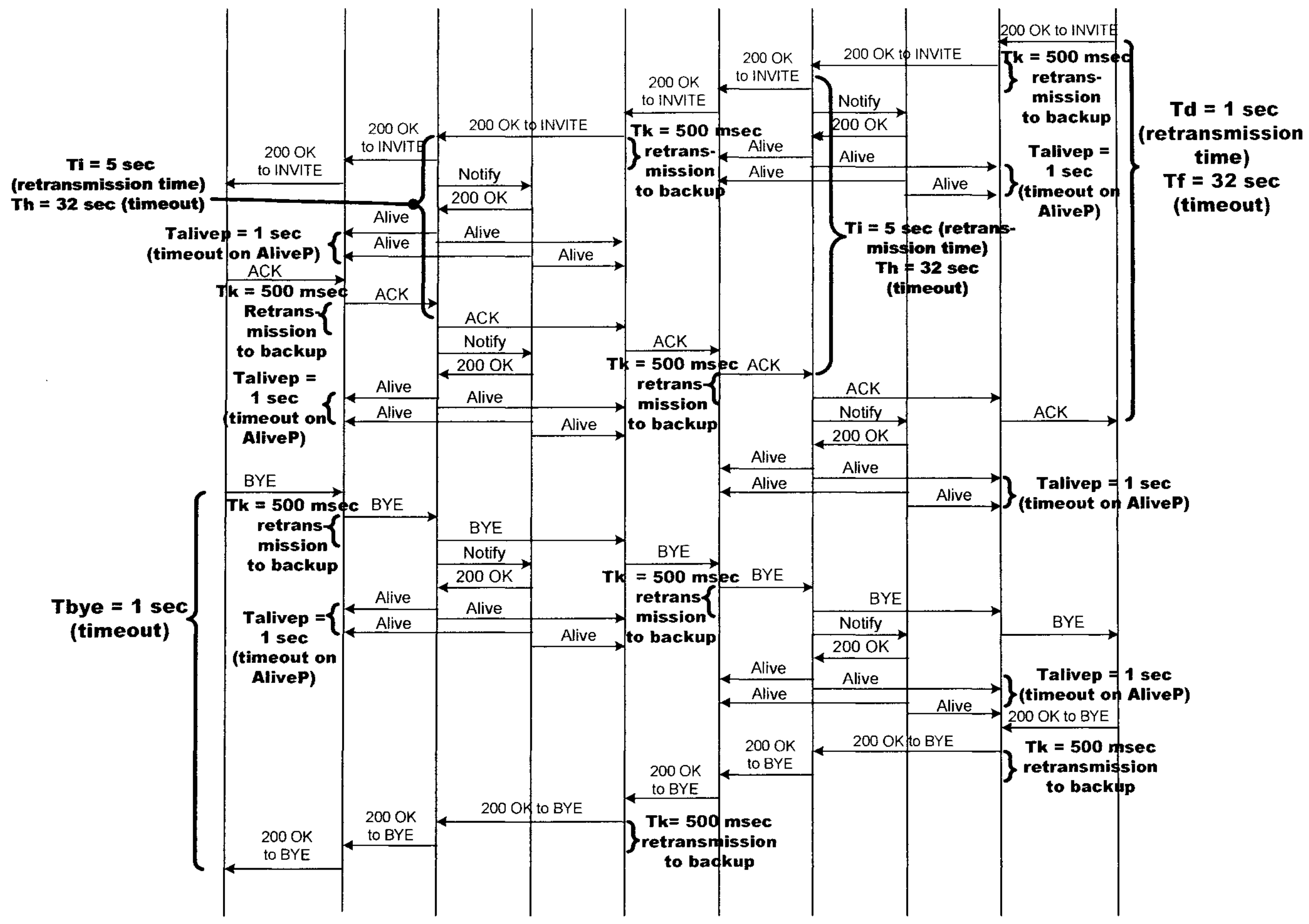

Ti = 5 sec (retransmission time) Th = 32 sec (timeout)
Talivep = 1 sec (timeout on AliveP)
Tk = 500 msec Retransmission to backup
Tbye = 1 sec (timeout)
200 OK to INVITE
ACK
BYE
Notify
Alive
200 OK
200 OK to BYE
Td = 1 sec (retransmission time) Tf = 32 sec (timeout)

Figure 4.35 SIP Message Exchange with Timing Constraints (Part 2)

4.6 Theoretical Performance Analysis

After analyzing the timing constraints placed on the messages within SIP it is evident that when creating a novel proposed approach to attain fault tolerance within SIP, several performance issues must be analyzed with respect to the number of messages which are introduced into the network as well the time required to carry out the SIP session establishment and termination between end users. With the addition of the Decider block, the number of messages within the network increases. The additional messages are *Alive, StateRequest, StateResponse, Subscribe* and *Notify*. These messages are sent over the network but only between the entities of the fault tolerance block (i.e. Decider server, primary proxy server and backup proxy server). The *Alive* messages are sent concurrently to the SIP requests and responses and therefore do not increase the time of a session. However, some additional network resources will be consumed for the state updating process between the primary and backup proxy servers.

The fault detection and recovery within our proposed approach is completed in less than 32 seconds which is the maximum timeout of the initial INVITE request. The alternative approaches for fault tolerance as described in Section 3.1 detect a fault, timeout, and restart the session. This implies that the 32 seconds are expired and then the steps to recovery are taken. In our approach, the faults are handled dynamically while the session is being established. The faults are handled at the moment of their occurrence except in the case when the primary proxy fails after receiving a message. In this case, the Decider will resend the message to the backup within 500 msec. Let us take the scenario where the INVITE and 180 RINGING messages are sent end to end. In the worst case, each server fails *after* receiving the messages and the total recovery time is 4 x (500 msec) + Tm (time to transmit a message) = 2 sec + Tm as compared to 32 + Tm in the other approaches. If there are additional proxy servers between the UA then also the time to recovery is significantly less than 32 sec. The timing analysis of our proposed approach in the case where the server fails at every point before the receipt of the provisional response is shown in Figure 4.36. Server failures at the other points produce the same timing scheme as the scenario shown.

Therefore, after theoretically analyzing our proposed approach, it can be stated that even though the number of messages has increased, the delay factor within the system is drastically reduced. This is summarized in Table 4.4

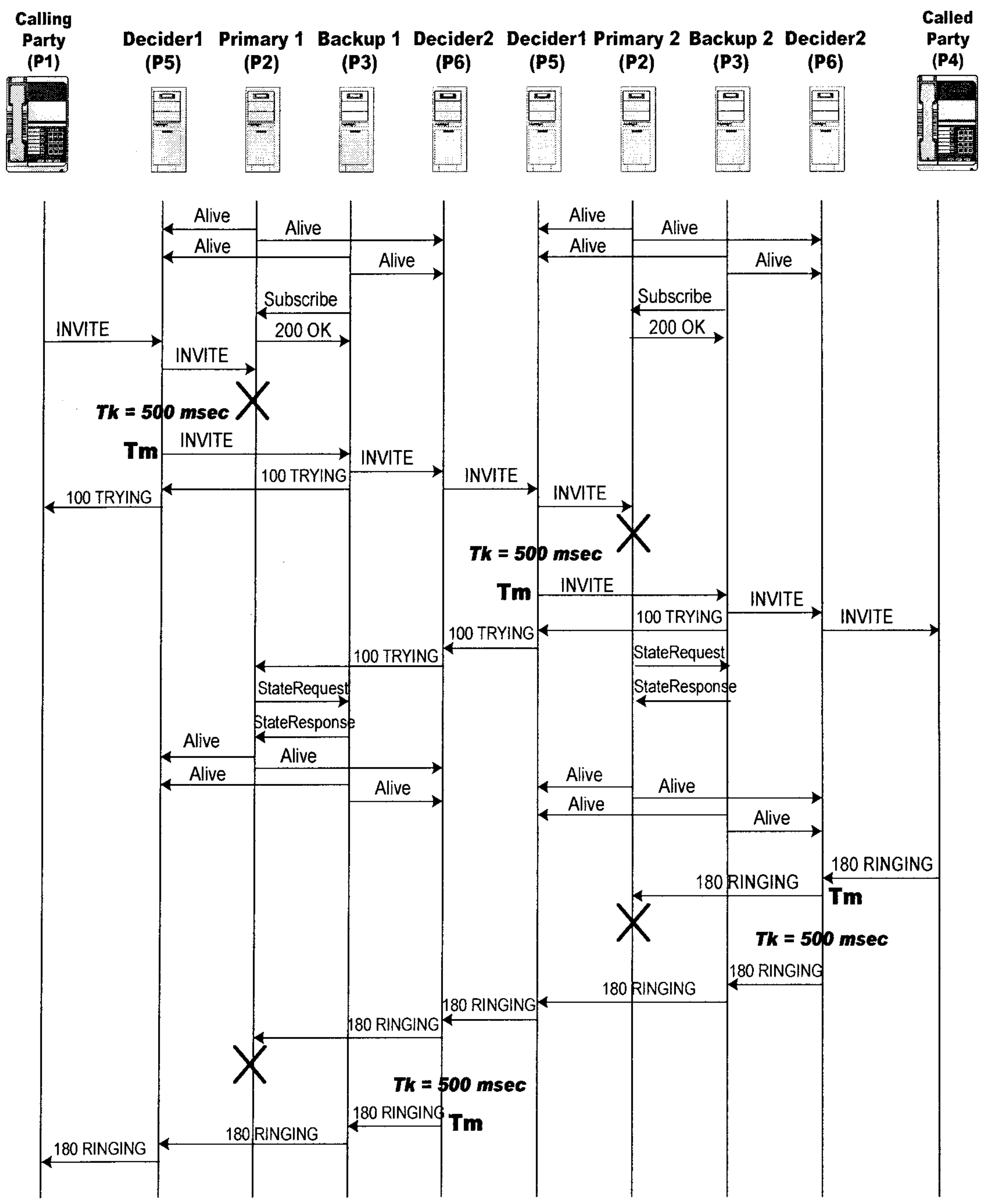


Figure 4.36 Worst Case Timing Analysis with the presence of faults

	Number of messages (Session Establishment using PRACK)	Time to Recovery (Worst Case)
Other Approaches	**7** [INVITE, 100 TRYING, 180 RINGING, 200 OK to INVITE, PRACK, 200 OK to PRACK, ACK]	**32 + Tm sec**
Proposed Approach	**12** [INVITE, 100 TRYING, 180 RINGING, 200 OK to INVITE, PRACK, 200 OK to PRACK, ACK, SUBSCRIBE, NOTIFY, ALIVE, STATEREQUEST, STATERESPONSE]	**12 + Tm sec**

Table 4.4 Summary of Performance Variables

4.7 Design Assumptions

Throughout the implementation of our proposed approach in SDL, there are several assumptions which we have taken. These allow the description of our fault tolerant Decider server solution to be complete without the implementation of the entire protocol.

As mentioned in Chapter 2, there are several requests which can be sent between end users. Among these, the CANCEL and OPTIONS requests were not implemented within the SDL model of our proposed approach. The CANCEL request will prematurely terminate the session and like all other messages, will flow through the proxy servers based on the decision of the Decider server. The CANCEL request does not require any retransmissions and therefore the complexity is reduced. The request can alter the status of the session however it is not essential when describing the failure and recovery process of the primary proxy server. Thus, this request is omitted from the implementation. The OPTIONS request is simply a query to determine the capabilities of another SIP entity and does not alter the session state. Therefore, it will be ignored if sent to a proxy server that has failed. The usage of this request has no added value with respect to our proposed approach and thus is eliminated from the implementation.

The next assumption which we have taken is with respect to the primary and backup proxy servers within the fault tolerance block. We assume that the servers are already registered with the Decider server. This prerequisite will allow the Decider server to forward the messages received from the end user agents. The registration process is not shown within the implementation as our focus was on attaining fault tolerance within the session establishment and termination phases of the SIP session.

Our implementation also uses the PRACK Extension to assure that reliable provisional responses are received from the Called Party. Recall from Chapter 2 that in order for this extension to be used, the Calling Party must contain 100rel in both the Supported and Require header fields of the INVITE request. Within our implementation, we are attempting to assure that the retransmissions are safely delivered even in the presence of a proxy failure and thus our final assumption is that the header fields within the messages are correctly set. This implies that the SIP entities involved in the session are aware and capable of handling this extension.

4.8 Summary

This chapter summarized the implementation which we have conducted for SIP inclusive of our proposed Decider server. This was done using the SDL modeling tool. This tool allows the definition of communication protocols by means of a hierarchy. The entire system inclusive of its communication with the environment was shown. The blocks and sub blocks which contained the workings of the individual SIP entities were shown and the processes and procedures defined for the SIP requests and responses were displayed. Finally, the workings of the processes were presented through the designed EFSM. The chapter continued with an overview of the timing constraints imposed by SIP and an analysis of the performance issues raised within the system design. The chapter concluded with the assumptions that we have taken during the implementation.

Chapter 5 Simulation and Verification

In this chapter, we describe the results obtained from the verification of our SDL model. We analyze the resulting Message Sequence Charts (MSC) from the SDL tool and verify if these match the expected results of the SIP fault scenarios introduced in Section 3.4.3. Finally, we discuss the limitations of the SDL tool and the manner in which our model was modified as a means of coping with these restrictions.

5.1 Fault Scenarios Analyzed

Several fault scenarios are analyzed against those theoretically proposed in Section 3.4.3. For each case, the backup server takes over the session when a fault occurs in the primary proxy server. Several points must be taken into consideration when analyzing the given MSC.

1) The message exchange for non-faulty scenarios are similar to those shown in Figure 3.8 and are thus eliminated from the MSC. These MSC are partial and display the occurrence of faults within the running session.
2) The transaction layer is eliminated from the MSC. The layer simply creates the request or response and transmits it to the calling or called party respectively. Since this also represents a non-faulty component of the session, it is eliminated.
3) Excluding the eleventh scenario, the backup proxy subscribes to the primary proxy at the beginning of the session prior to the issue of an initial INVITE Request. For every successful non-faulty transfer of a message (i.e. any request or response), the

primary issues a notification message to the backup proxy thereby maintaining both servers at a consistent state.

4) Periodically, the primary and backup proxies issue *Alive* messages to the Decider blocks to inform their current status (i.e. alive or not).
5) The MSC begins with a fault occurrence simulated with the *StopP* message fed from the environment into the system. The faults considered are not permanent since recovery of the primary proxy server can occur at any point although the first nine scenarios do not show any recovery within the message flow.
6) Faults may occur in one or more primary proxy servers. Within our model, two proxy servers exist and either primary may fail. In several of our scenarios, both primary proxies fail in order to illustrate that faults can be tolerated in more than one primary proxy at a time. In other scenarios, a single proxy fails which shows that redundancy can be localized to a particular server.
7) After a primary proxy server has failed, it moves to the idle state and all messages sent to it are ignored. The absorption of these messages is simulated by taking the request or response as an input and then returning to the idle state. No change of state occurs until the primary proxy server recovers.

The first scenario shows the occurrence of a fault within both primary proxy servers at the beginning of a session. Since no *AliveP* message is received from the primary proxy server, the Decider block detects a fault and thus the *D1_Process* sends the INVITE Message to the backup server. The session continues through the backup proxy without the primary server's involvement. The resulting MSC is shown in Figure 5.1.

In the second scenario, the primary proxy server fails before receiving the Provisional Response from the Called Party. Since no *AliveP* message is received from the primary proxy server, the Decider block detects a fault and thus the *D2_Process* sends the 180 RINGING Response to the backup server. As expected all subsequent messages are routed through the backup proxy server. The resulting MSC is shown in Figure 5.2.

In the third scenario, the primary proxy server fails before receiving the Error Response from the Called Party. Since no *AliveP* message is received from the primary proxy server, the Decider block detects a fault and thus the *D2_Process* sends the Error Response to the backup server. As expected all subsequent messages are routed through the backup proxy server. The resulting MSC is shown in Figure 5.3.

In the fourth scenario, the primary proxy server fails before receiving PRACK from the Calling Party. Since no *AliveP* message is received from the primary proxy server, the Decider block detects a fault and thus the *D1_Process* sends the PRACK Request to the backup server. As expected all subsequent messages are routed through the backup proxy server. The resulting MSC is shown in Figure 5.4.

In the fifth scenario, the primary proxy server fails before receiving 200 OK to PRACK from the Called Party. Since no *AliveP* message is received from the primary proxy server, the Decider block detects a fault and thus the *D2_Process* sends the 200 OK to PRACK Response to the backup server. As expected all subsequent messages are routed through the backup proxy server. The resulting MSC is shown in Figure 5.5.

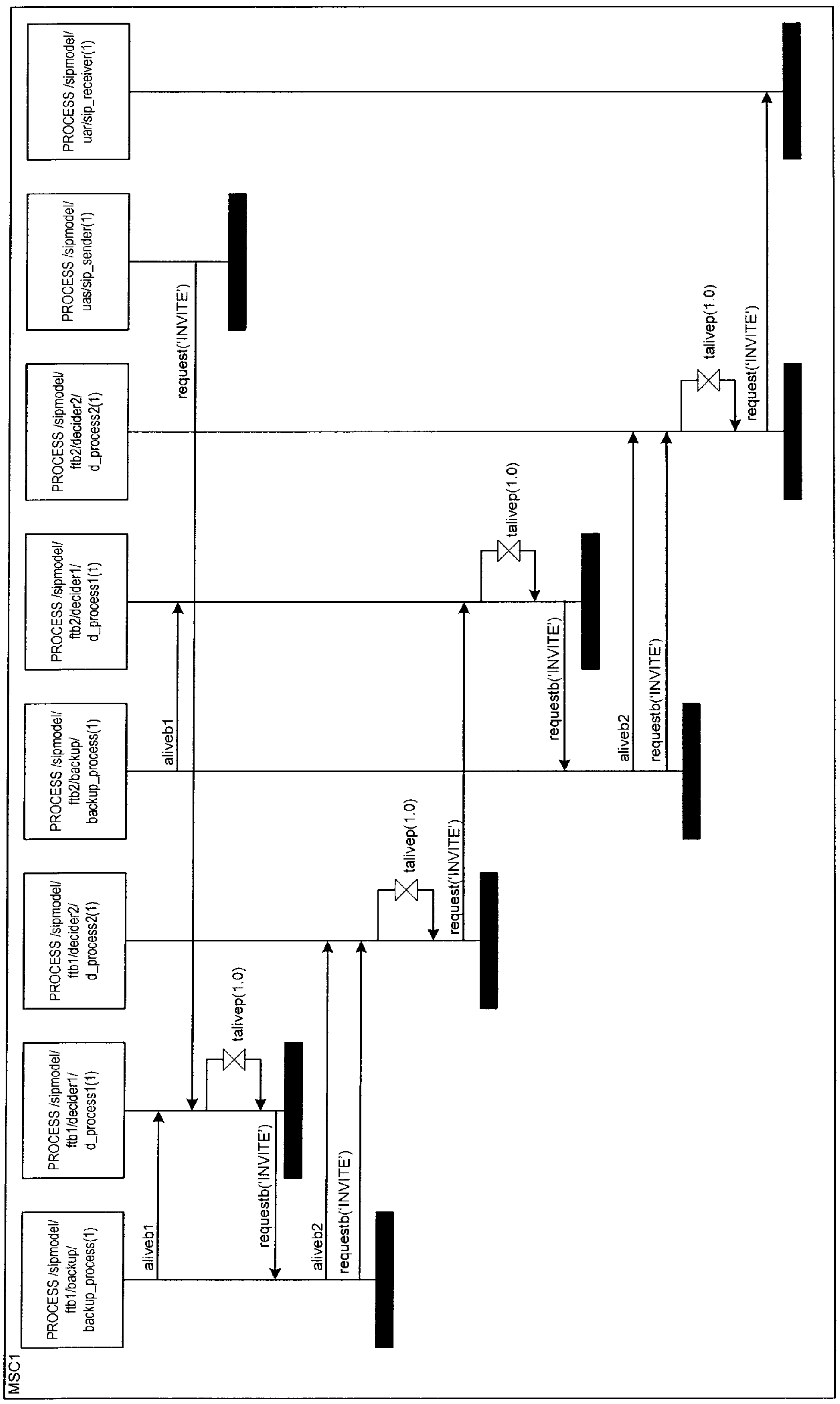

MSC1
PROCESS /sipmodel/ ftb1/backup/ backup_process(1)
PROCESS /sipmodel/ ftb1/decider1/ d_process1(1)
PROCESS /sipmodel/ ftb1/decider2/ d_process2(1)
PROCESS /sipmodel/ ftb2/backup/ backup_process(1)
PROCESS /sipmodel/ ftb2/decider1/ d_process1(1)
PROCESS /sipmodel/ ftb2/decider2/ d_process2(1)
PROCESS /sipmodel/ uas/sip_sender(1)
PROCESS /sipmodel/ uar/sip_receiver(1)
request('INVITE')
aliveb1
talivep(1.0)
requestb('INVITE')
aliveb2
requestb('INVITE')
talivep(1.0)
request('INVITE')
aliveb1
talivep(1.0)
requestb('INVITE')
aliveb2
requestb('INVITE')
talivep(1.0)
request('INVITE')

Figure 5.1 MSC of primary proxy failure at the beginning of the session

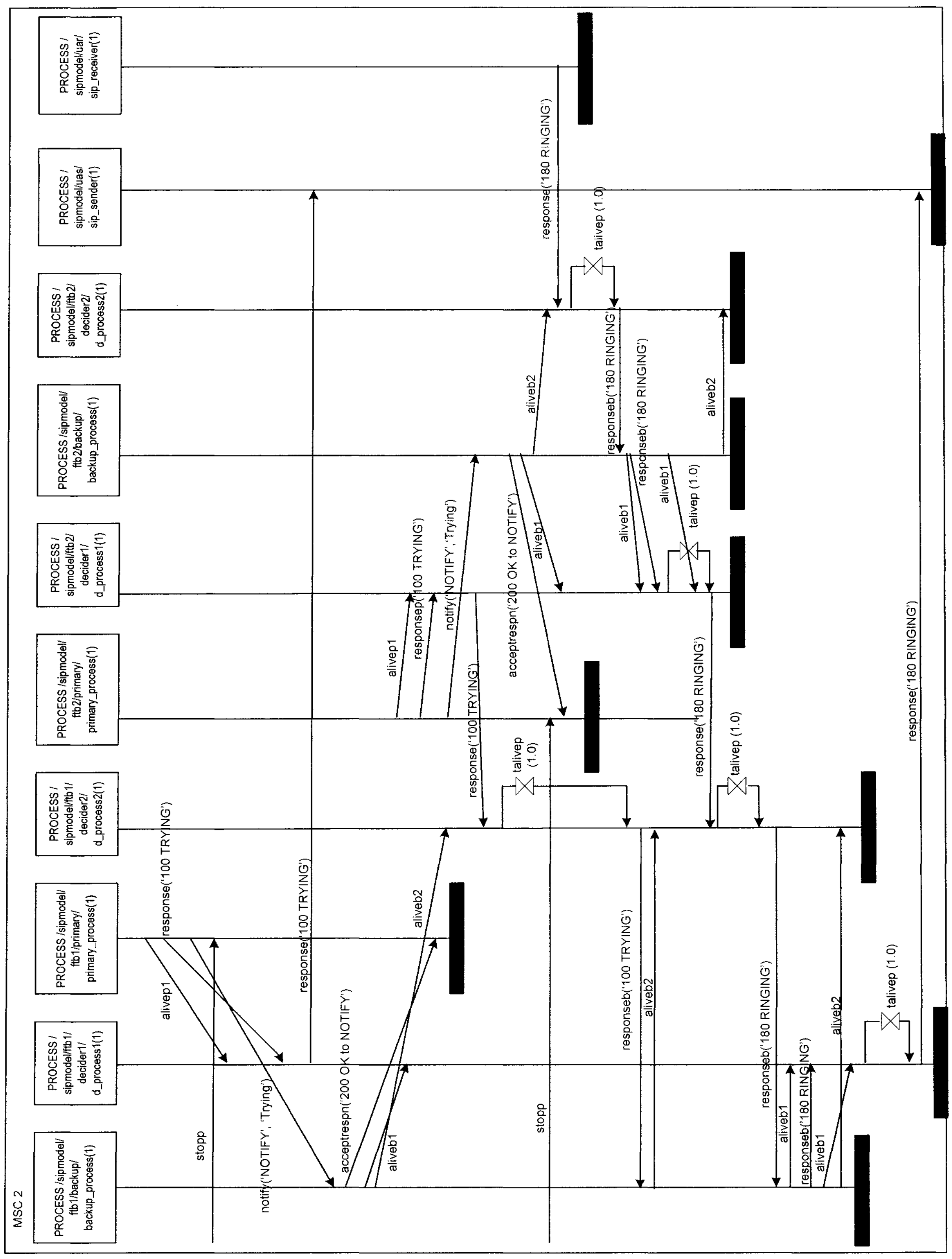

MSC 2
PROCESS /sipmodel/ ftb1/backup/ backup_process(1)
PROCESS / sipmodel/ftb1/ decider1/ d_process1(1)
PROCESS /sipmodel/ ftb1/primary/ primary_process(1)
PROCESS / sipmodel/ftb1/ decider2/ d_process2(1)
PROCESS /sipmodel/ ftb2/primary/ primary_process(1)
PROCESS / sipmodel/ftb2/ decider1/ d_process1(1)
PROCESS /sipmodel/ ftb2/backup/ backup_process(1)
PROCESS / sipmodel/ftb2/ decider2/ d_process2(1)
PROCESS / sipmodel/uas/ sip_sender(1)
PROCESS / sipmodel/uar/ sip_receiver(1)
stopp
alivep1
response('100 TRYING')
notify('NOTIFY', 'Trying')
response('100 TRYING')
acceptrespn('200 OK to NOTIFY')
aliveb1
aliveb2
alivep1
responsep('100 TRYING')
notify('NOTIFY', 'Trying')
response('100 TRYING')
talivep (1.0)
acceptrespn('200 OK to NOTIFY')
aliveb1
stopp
aliveb2
response('180 RINGING')
talivep (1.0)
responseb('180 RINGING')
responseb('180 RINGING')
aliveb1
aliveb1
talivep (1.0)
aliveb2
response('180 RINGING')
talivep (1.0)
responseb('100 TRYING')
aliveb2
responseb('180 RINGING')
aliveb1
responseb('180 RINGING')
aliveb1
aliveb2
talivep (1.0)
response('180 RINGING')

Figure 5.2 MSC of primary proxy failure before receiving 180 Ringing

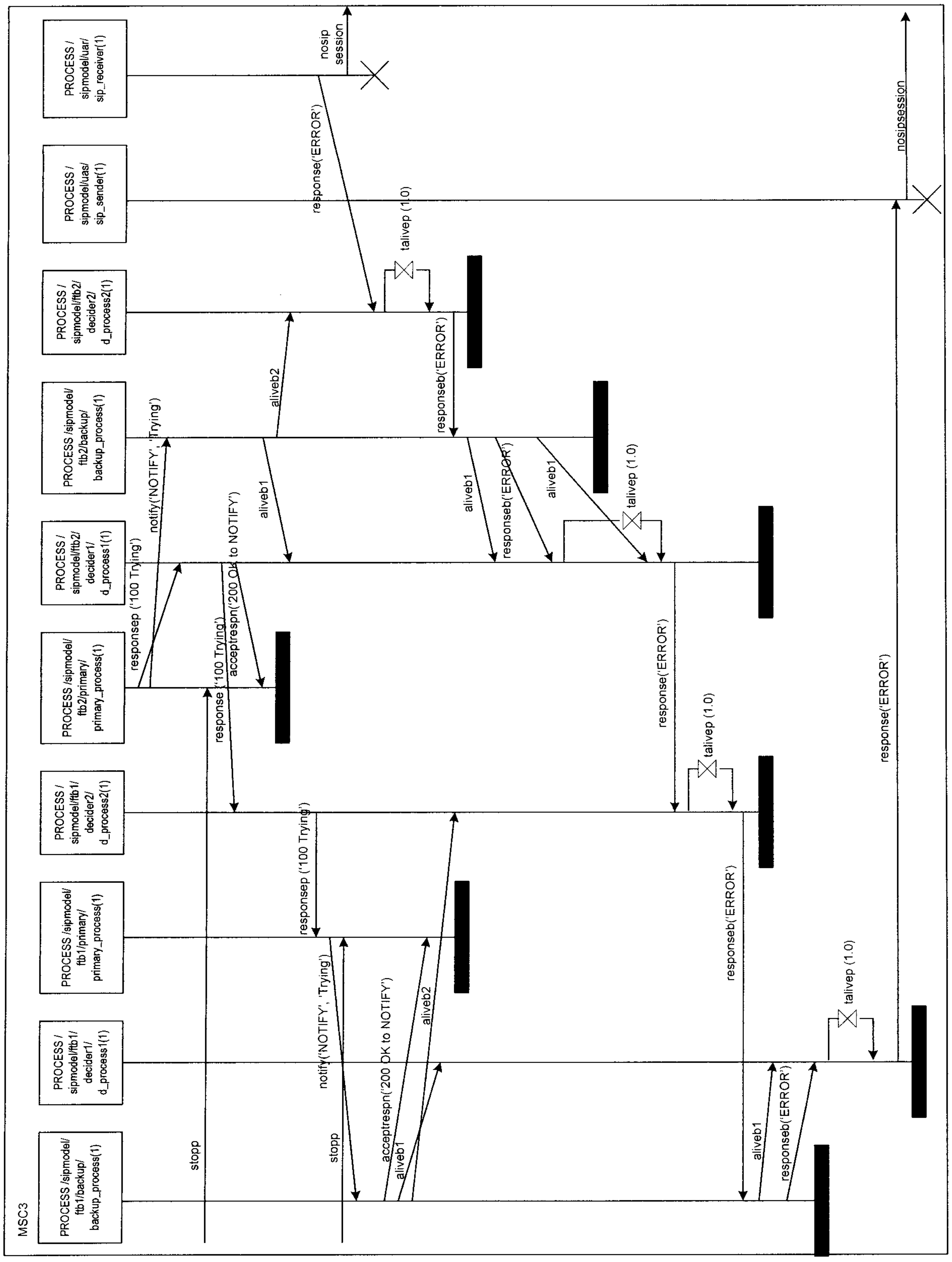

MSC3
PROCESS /sipmodel/ftb1/backup/backup_process(1)
PROCESS / sipmodel/ftb1/ decider1/ d_process1(1)
PROCESS /sipmodel/ ftb1/primary/ primary_process(1)
PROCESS / sipmodel/ftb1/ decider2/ d_process2(1)
PROCESS /sipmodel/ ftb2/primary/ primary_process(1)
PROCESS / sipmodel/ftb2/ decider1/ d_process1(1)
PROCESS /sipmodel/ ftb2/backup/ backup_process(1)
PROCESS / sipmodel/ftb2/ decider2/ d_process2(1)
PROCESS / sipmodel/uas/ sip_sender(1)
PROCESS / sipmodel/uar/ sip_receiver(1)
nosip session
response('ERROR')
talivep (1.0)
responseb('ERROR')
aliveb2
notify('NOTIFY', 'Trying')
aliveb1
responseb('ERROR')
aliveb1
talivep (1.0)
responsep ('100 Trying')
acceptrespn('200 OK to NOTIFY')
response ('100 Trying')
response('ERROR')
talivep (1.0)
response('ERROR')
nosipsession
stopp
responsep ('100 Trying')
notify('NOTIFY', 'Trying')
acceptrespn('200 OK to NOTIFY')
aliveb2
stopp
aliveb1
responseb('ERROR')
aliveb1
responseb('ERROR')
talivep (1.0)

Figure 5.3 MSC of primary proxy failure before receiving Error Response

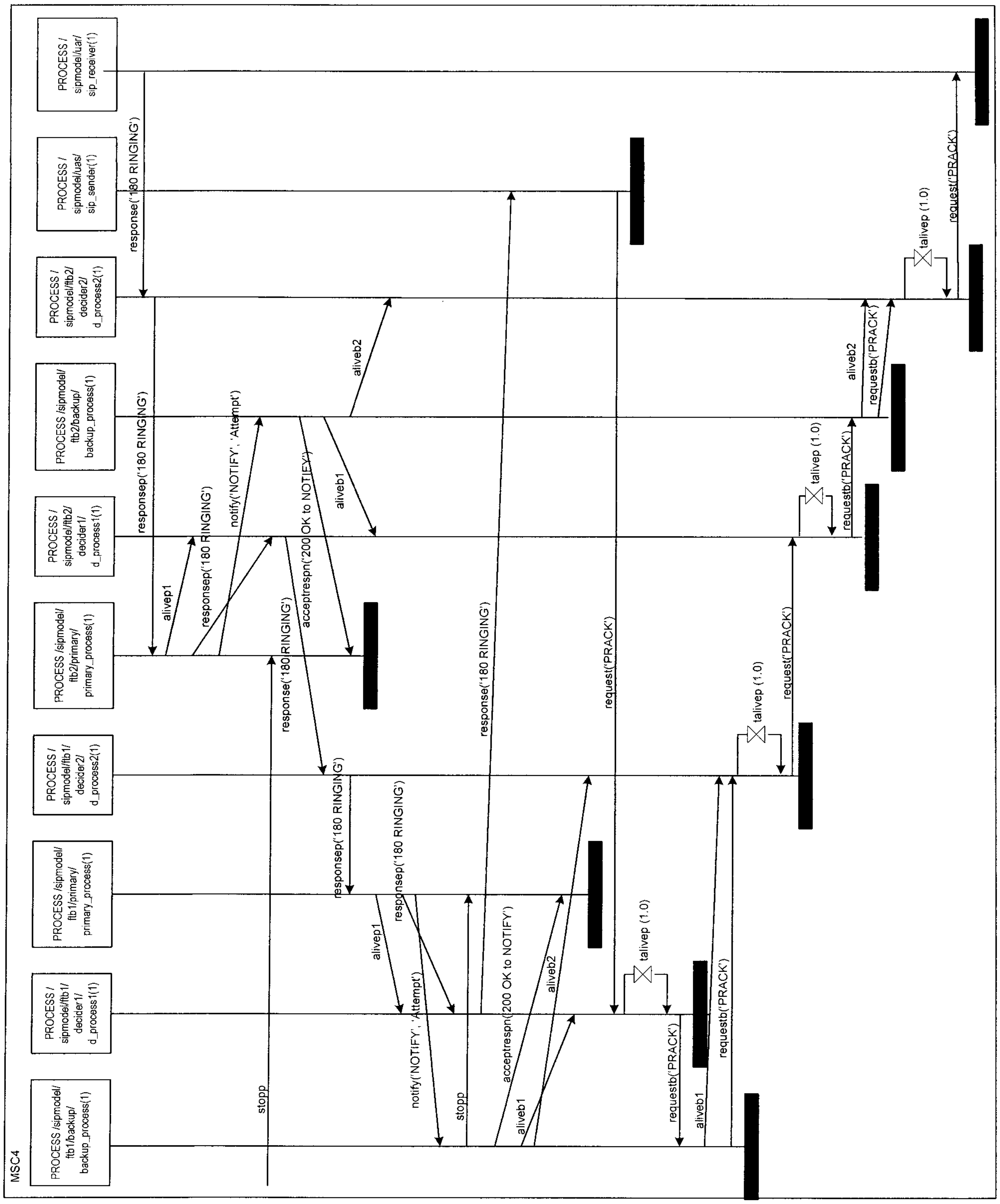

MSC4
PROCESS /sipmodel/ftb1/backup/backup_process(1)
PROCESS / sipmodel/ftb1/decider1/d_process1(1)
PROCESS /sipmodel/ftb1/primary/primary_process(1)
PROCESS / sipmodel/ftb1/decider2/d_process2(1)
PROCESS /sipmodel/ftb2/primary/primary_process(1)
PROCESS / sipmodel/ftb2/decider1/d_process1(1)
PROCESS /sipmodel/ftb2/backup/backup_process(1)
PROCESS / sipmodel/ftb2/decider2/d_process2(1)
PROCESS / sipmodel/uas/sip_sender(1)
PROCESS / sipmodel/uar/sip_receiver(1)
response('180 RINGING')
responsep('180 RINGING')
alivep1
responsep('180 RINGING')
notify('NOTIFY', 'Attempt')
aliveb2
aliveb1
acceptrespn('200 OK to NOTIFY')
response('180 RINGING')
stopp
responsep('180 RINGING')
alivep1
responsep('180 RINGING')
notify('NOTIFY', 'Attempt')
stopp
acceptrespn('200 OK to NOTIFY')
aliveb1
aliveb2
response('180 RINGING')
request('PRACK')
talivep (1.0)
requestb('PRACK')
aliveb1
requestb('PRACK')
talivep (1.0)
request('PRACK')
talivep (1.0)
requestb('PRACK')
aliveb2
requestb('PRACK')
talivep (1.0)
request('PRACK')

Figure 5.4 MSC of primary proxy failure before receiving PRACK

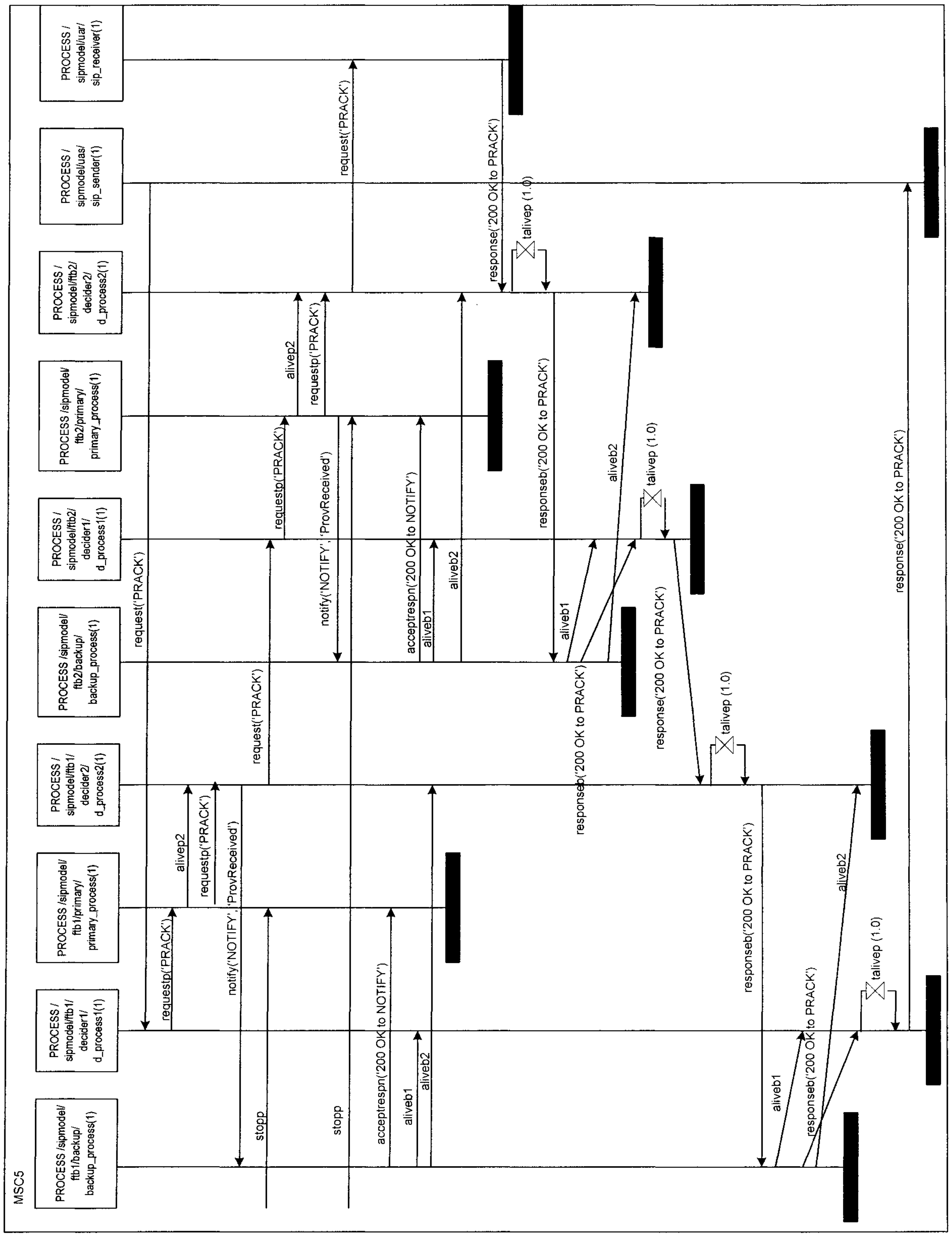

MSC5
PROCESS /sipmodel/uac/sip_receiver(1)
PROCESS /sipmodel/uas/sip_sender(1)
PROCESS /sipmodel/ftb2/decider2/d_process2(1)
PROCESS /sipmodel/ftb2/primary/primary_process(1)
PROCESS /sipmodel/ftb2/decider1/d_process1(1)
PROCESS /sipmodel/ftb2/backup/backup_process(1)
PROCESS /sipmodel/ftb1/decider2/d_process2(1)
PROCESS /sipmodel/ftb1/primary/primary_process(1)
PROCESS /sipmodel/ftb1/decider1/d_process1(1)
PROCESS /sipmodel/ftb1/backup/backup_process(1)
request('PRACK')
requestp('PRACK')
alivep2
notify('NOTIFY', 'ProvReceived')
stopp
acceptrespn('200 OK to NOTIFY')
aliveb1
aliveb2
response('200 OK to PRACK')
talivep (1.0)
responseb('200 OK to PRACK')

Figure 5.5 MSC of primary proxy failure before receiving 200 OK to PRACK

In the sixth scenario, the primary proxy server fails before receiving 200 OK to INVITE from the Called Party. Since no *AliveP* message is received from the primary proxy server, the Decider block detects a fault and thus the *D2_Process* sends the 200 OK to INVITE Response to the backup server. As expected all subsequent messages are routed through the backup proxy server. The resulting MSC is shown in Figure 5.6.

In the seventh scenario, the primary proxy server fails before receiving BYE from the Calling Party. As expected the response is routed through the backup proxy server. The resulting MSC is shown in Figure 5.7.

In the eighth scenario, the primary proxy server fails after sending an *AliveP* message to the Decider. Since an *AliveP* message is received from the primary proxy server, the Decider block does not detect a fault and thus the *D1_Process* sends the 180 RINGING Request to the primary server. The 180 RINGING Request however is not transmitted by the proxy. The Decider internally times out and reissues the saved message to the backup proxy server. As expected all subsequent messages are routed through the backup proxy server. The resulting MSC is shown in Figure 5.8.

In the ninth scenario, the second primary proxy server fails after receiving the 200 OK to PRACK message from the Called Party. At this point, the message which was previously stored into the first buffer is resent to the backup server. The backup takes over the session until the primary proxy server recovers. Note that if the 200 OK to INVITE message is issued by the Called Party before the retransmission of the 200 OK to PRACK message then it is stored in a second buffer and sent later. This saving is required since the message issuance of the Called party cannot be controlled. The resulting MSC is shown in Figure 5.9.

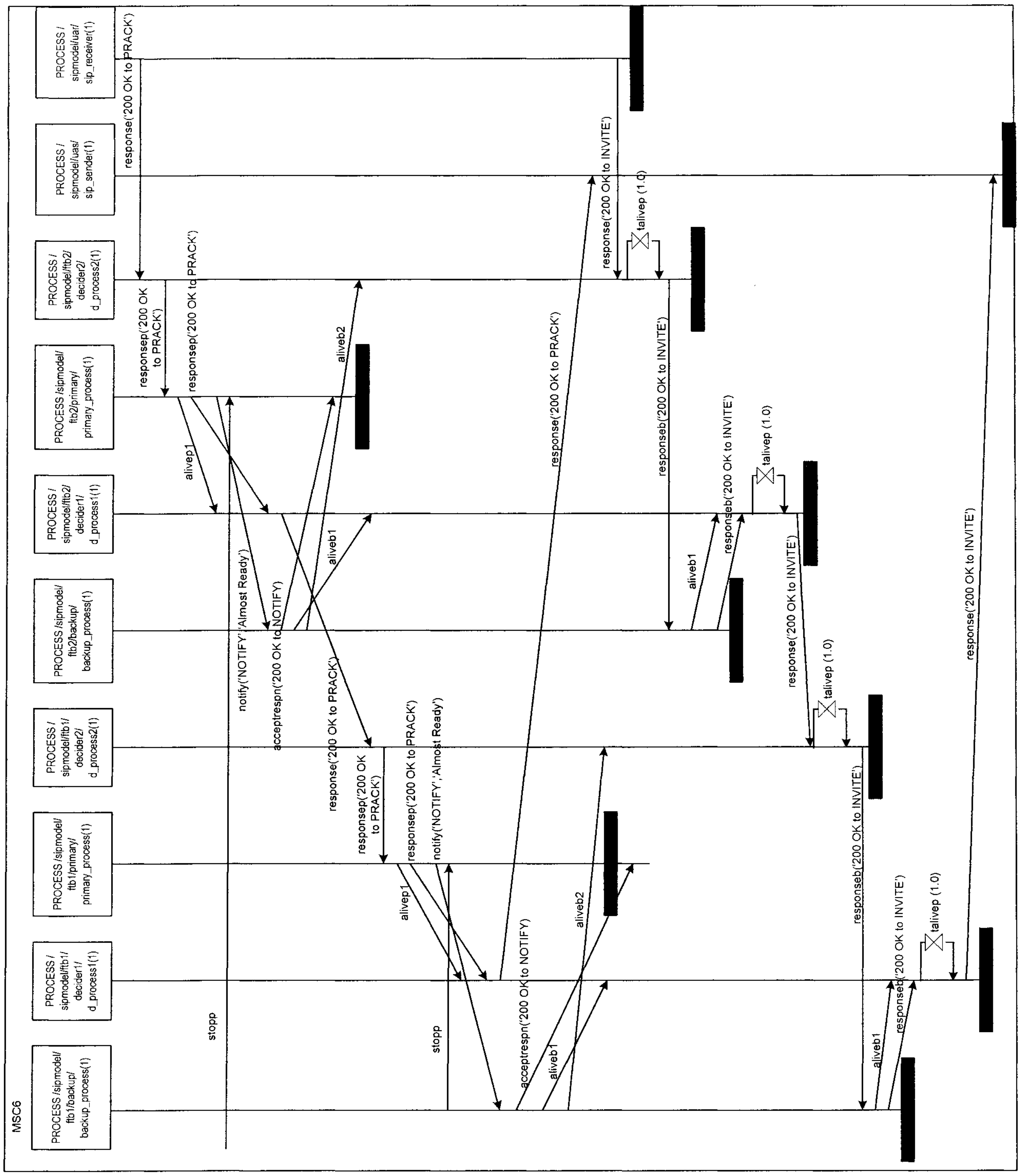

MSC6
PROCESS /sipmodel/ ftb1/backup/ backup_process(1)
PROCESS / sipmodel/ftb1/ decider1/ d_process1(1)
PROCESS /sipmodel/ ftb1/primary/ primary_process(1)
PROCESS / sipmodel/ftb1/ decider2/ d_process2(1)
PROCESS /sipmodel/ ftb2/backup/ backup_process(1)
PROCESS / sipmodel/ftb2/ decider1/ d_process1(1)
PROCESS /sipmodel/ ftb2/primary/ primary_process(1)
PROCESS / sipmodel/ftb2/ decider2/ d_process2(1)
PROCESS / sipmodel/uas/ sip_sender(1)
PROCESS / sipmodel/uar/ sip_receiver(1)
response('200 OK to PRACK')
responsep('200 OK to PRACK')
responsep('200 OK to PRACK')
alivep1
alive b2
notify('NOTIFY','Almost Ready')
acceptrespn('200 OK to NOTIFY)
aliveb1
response('200 OK to PRACK')
stopp
responsep('200 OK to PRACK')
responsep('200 OK to PRACK')
notify('NOTIFY','Almost Ready')
alivep1
stopp
acceptrespn('200 OK to NOTIFY)
aliveb1
aliveb2
response('200 OK to PRACK')
response('200 OK to INVITE')
talivep (1.0)
responseb('200 OK to INVITE')
aliveb1
responseb('200 OK to INVITE')
talivep (1.0)
response('200 OK to INVITE')
talivep (1.0)
responseb('200 OK to INVITE')
aliveb1
responseb('200 OK to INVITE')
talivep (1.0)
response('200 OK to INVITE')

Figure 5.6 MSC of primary proxy failure before receiving 200 OK to INVITE

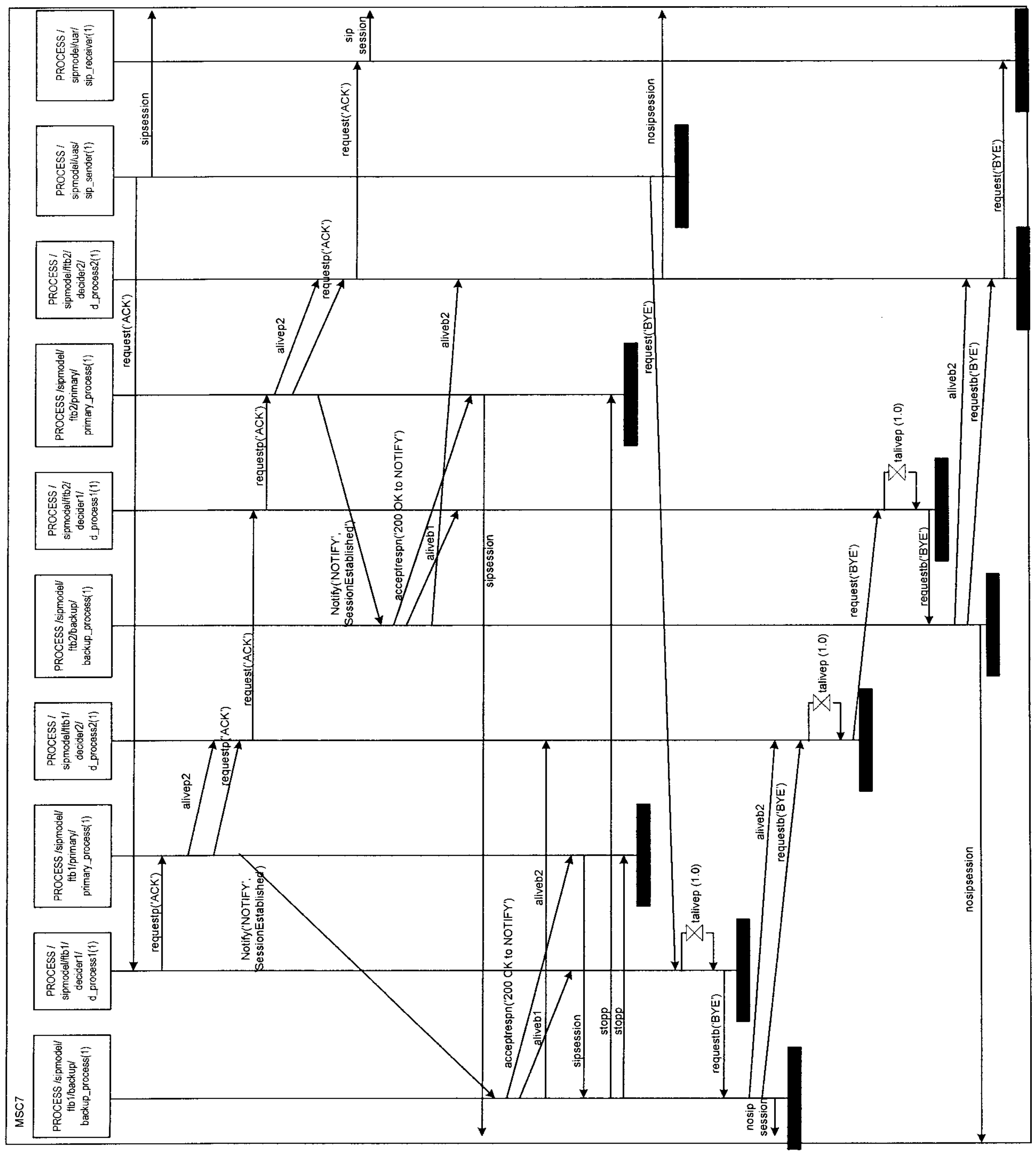

MSC7
PROCESS /sipmodel/ftb1/backup/backup_process(1)
PROCESS /sipmodel/ftb1/decider1/d_process1(1)
PROCESS /sipmodel/ftb1/primary/primary_process(1)
PROCESS /sipmodel/ftb1/decider2/d_process2(1)
PROCESS /sipmodel/ftb2/backup/backup_process(1)
PROCESS /sipmodel/ftb2/decider1/d_process1(1)
PROCESS /sipmodel/ftb2/primary/primary_process(1)
PROCESS /sipmodel/ftb2/decider2/d_process2(1)
PROCESS /sipmodel/uas/sip_sender(1)
PROCESS /sipmodel/uar/sip_receiver(1)
request('ACK')
sipsession
requestp('ACK')
alivep2
Notify('NOTIFY', 'SessionEstablished')
request('ACK')
acceptrespn('200 OK to NOTIFY')
aliveb1
aliveb2
sipsession
stopp
talivep (1.0)
requestb('BYE')
request('BYE')
nosipsession
nosip session
sip session

Figure 5.7 MSC of primary proxy failure before receiving BYE

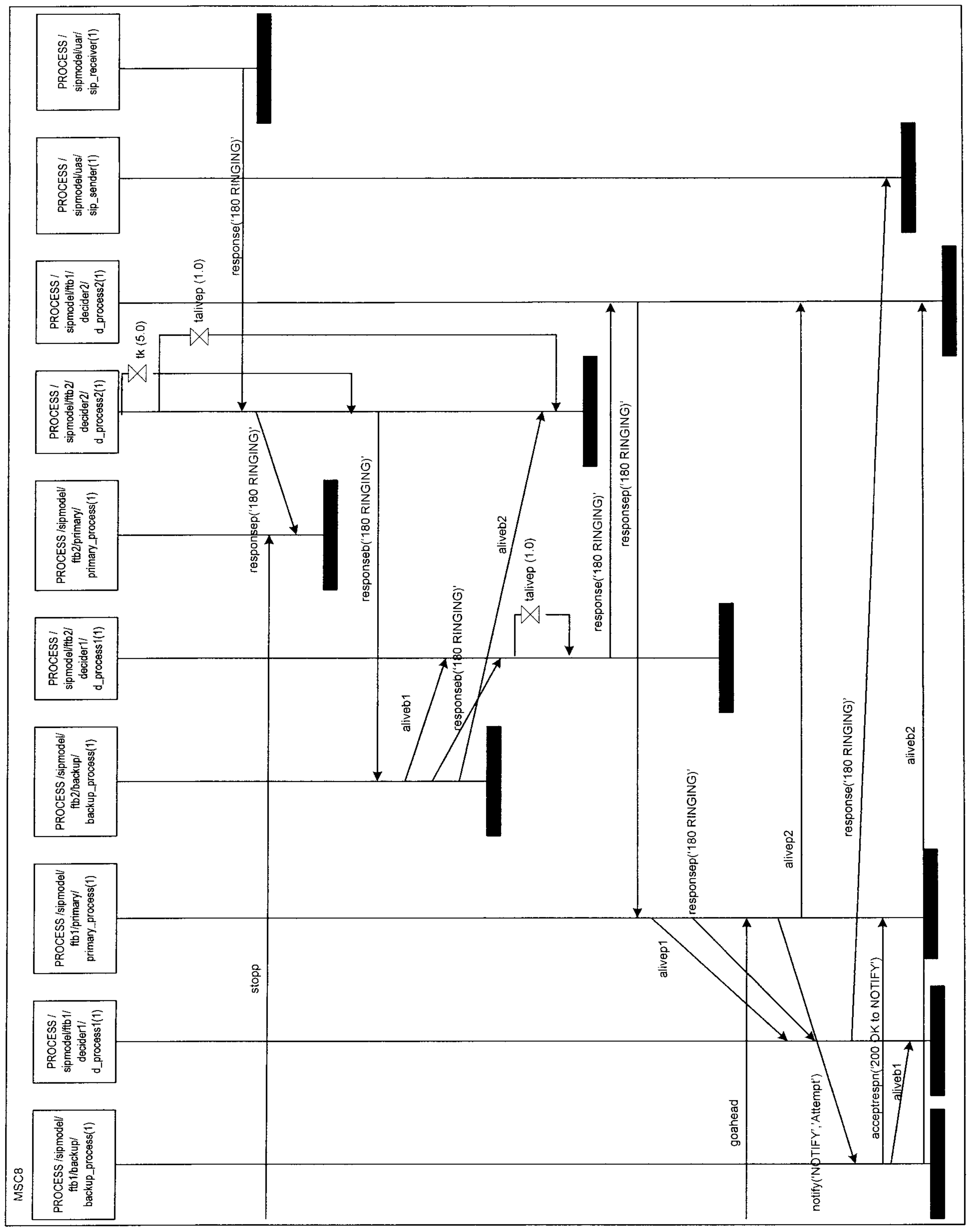
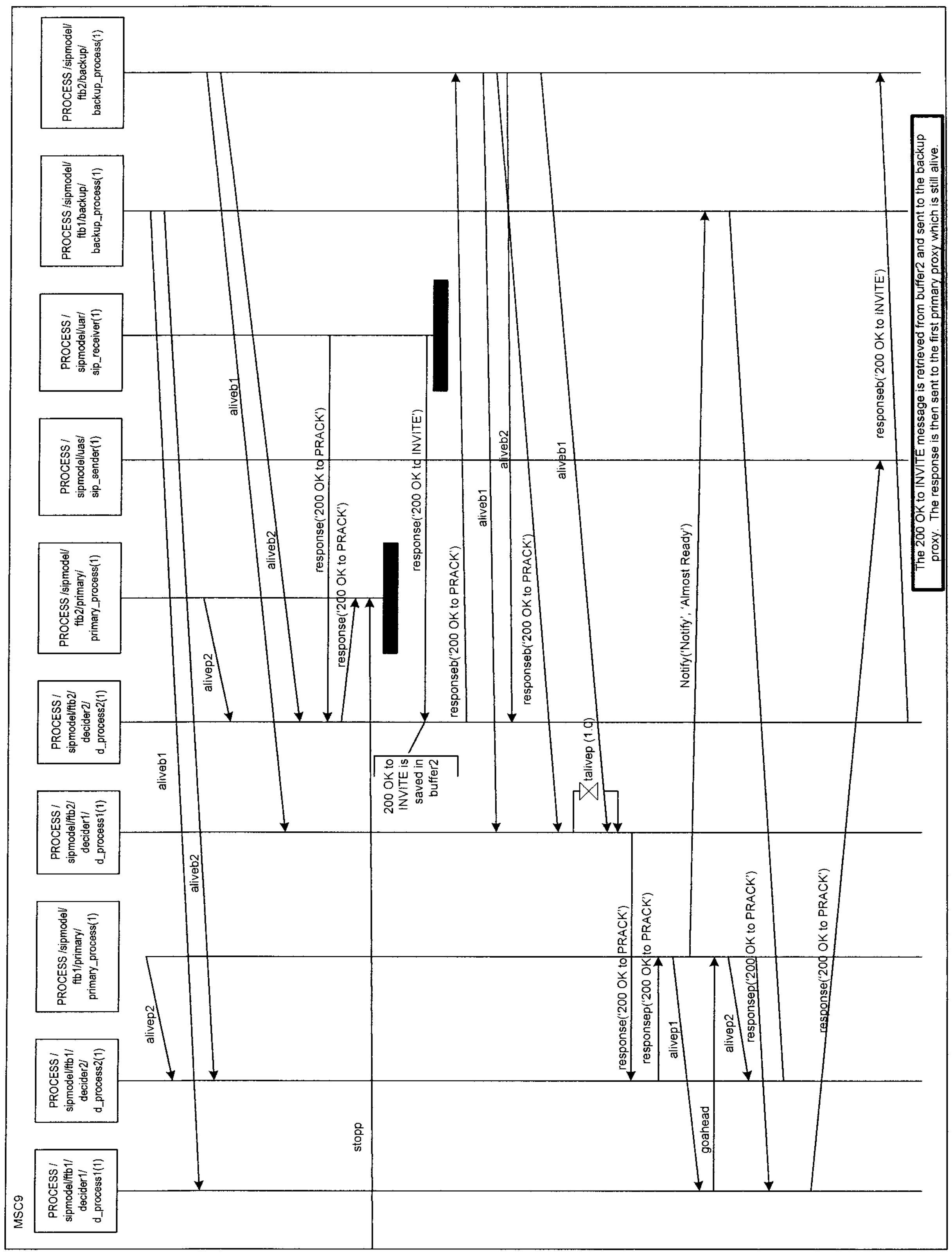

MSC8
PROCESS /sipmodel/ ftb1/backup/ backup_process(1)
PROCESS / sipmodel/ftb1/ decider1/ d_process1(1)
PROCESS /sipmodel/ ftb1/primary/ primary_process(1)
PROCESS /sipmodel/ ftb2/backup/ backup_process(1)
PROCESS / sipmodel/ftb2/ decider1/ d_process1(1)
PROCESS /sipmodel/ ftb2/primary/ primary_process(1)
PROCESS / sipmodel/ftb2/ decider2/ d_process2(1)
PROCESS / sipmodel/ftb1/ decider2/ d_process2(1)
PROCESS / sipmodel/uas/ sip_sender(1)
PROCESS / sipmodel/uar/ sip_receiver(1)
response('180 RINGING)'
tk (5.0)
talivep (1.0)
responsep('180 RINGING)'
responseb('180 RINGING)'
aliveb2
aliveb1
responseb('180 RINGING)'
talivep (1.0)
response('180 RINGING)'
responsep('180 RINGING)'
stopp
alivep1
responsep('180 RINGING)'
alivep2
response('180 RINGING)'
aliveb2
goahead
notify('NOTIFY','Attempt')
acceptrespn('200 OK to NOTIFY')
aliveb1

Figure 5.8 MSC of primary proxy failure after receiving 180 RINGING



MSC9
PROCESS / sipmodel/ftb1/ decider1/ d_process1(1)
PROCESS / sipmodel/ftb1/ decider2/ d_process2(1)
PROCESS /sipmodel/ ftb1/primary/ primary_process(1)
PROCESS / sipmodel/ftb2/ decider1/ d_process1(1)
PROCESS / sipmodel/ftb2/ decider2/ d_process2(1)
PROCESS /sipmodel/ ftb2/primary/ primary_process(1)
PROCESS / sipmodel/uas/ sip_sender(1)
PROCESS / sipmodel/uar/ sip_receiver(1)
PROCESS /sipmodel/ ftb1/backup/ backup_process(1)
PROCESS /sipmodel/ ftb2/backup/ backup_process(1)
alivep2
aliveb1
aliveb2
alivep2
aliveb1
aliveb2
response('200 OK to PRACK')
response('200 OK to PRACK')
stopp
response('200 OK to INVITE')
200 OK to INVITE is saved in buffer2
responseb('200 OK to PRACK')
aliveb1
aliveb2
responseb('200 OK to PRACK')
aliveb1
talivep (1.0)
response('200 OK to PRACK')
responsep('200 OK to PRACK')
alivep1
goahead
Notify('Notify', 'Almost Ready')
alivep2
responsep('200 OK to PRACK')
response('200 OK to PRACK')
responseb('200 OK to INVITE')
The 200 OK to INVITE message is retrieved from buffer2 and sent to the backup proxy. The response is then sent to the first primary proxy which is still alive.

Figure 5.9 MSC of primary proxy failure after receiving 200 OK to PRACK

In the tenth scenario, the primary proxy server fails before receiving the INVITE Request from the Calling Party. As expected the response is routed through the backup proxy server. After the issue of the 180 RINGING response, the server recovers. The primary proxy server requests the state of the session from the backup proxy server, issues an *AliveP* message to the Decider Block and resumes operation within the session. The resulting MSC is shown in Figure 5.10.

In the eleventh scenario, the backup proxy server subscribes to the primary proxy server after the INVITE Request has been sent by the Calling Party. As expected, no notification messages are issued prior to the subscription. Once the subscription is accepted by the primary proxy server, notification messages will be sent to the backup. The resulting MSC is shown in Figure 5.11.

After analysis of these MSC, we have verified that our proposed approach handles the occurrence of faults anywhere within the SIP session. Furthermore, the recovery of the primary server should allow it to resume its role within the running session. This criterion is also fulfilled since the current state is requested from the backup server and then the *Alive* messages inform the deciding block of the availability of the server. Further, the backup should be allowed to subscribe to the primary at any stage within the session. This criterion is also fulfilled within our design. Finally the finite state machines should be free of any deadlock and/or livelock scenarios. These properties were tested within the SDL tool by running and analyzing the derived MSC.

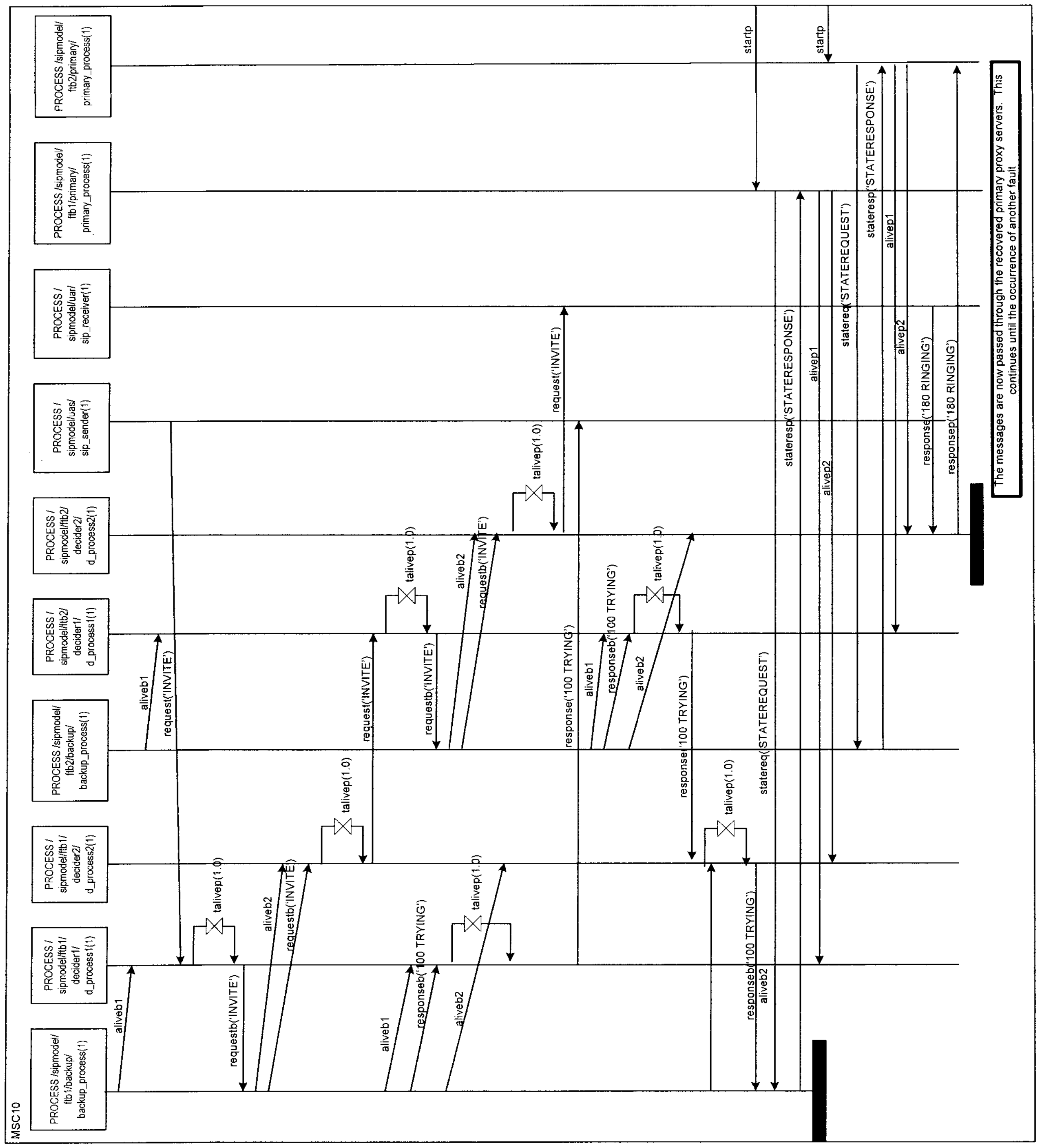

MSC10
PROCESS /sipmodel/ftb1/backup/backup_process(1)
PROCESS / sipmodel/ftb1/decider1/d_process1(1)
PROCESS / sipmodel/ftb1/decider2/d_process2(1)
PROCESS /sipmodel/ftb2/backup/backup_process(1)
PROCESS / sipmodel/ftb2/decider1/d_process1(1)
PROCESS / sipmodel/ftb2/decider2/d_process2(1)
PROCESS / sipmodel/uas/sip_sender(1)
PROCESS / sipmodel/uar/sip_receiver(1)
PROCESS /sipmodel/ftb1/primary/primary_process(1)
PROCESS /sipmodel/ftb2/primary/primary_process(1)
aliveb1
request('INVITE')
talivep(1.0)
requestb('INVITE')
aliveb2
requestb('INVITE')
responseb('100 TRYING')
response('100 TRYING')
statereq('STATEREQUEST')
stateresp('STATERESPONSE')
alivep1
alivep2
startp
response('180 RINGING')
responsep('180 RINGING')
The messages are now passed through the recovered primary proxy servers. This continues until the occurrence of another fault

Figure 5.10 MSC of primary proxy failure before receiving INVITE and Proxy Recovery before receipt of 180 RINGING

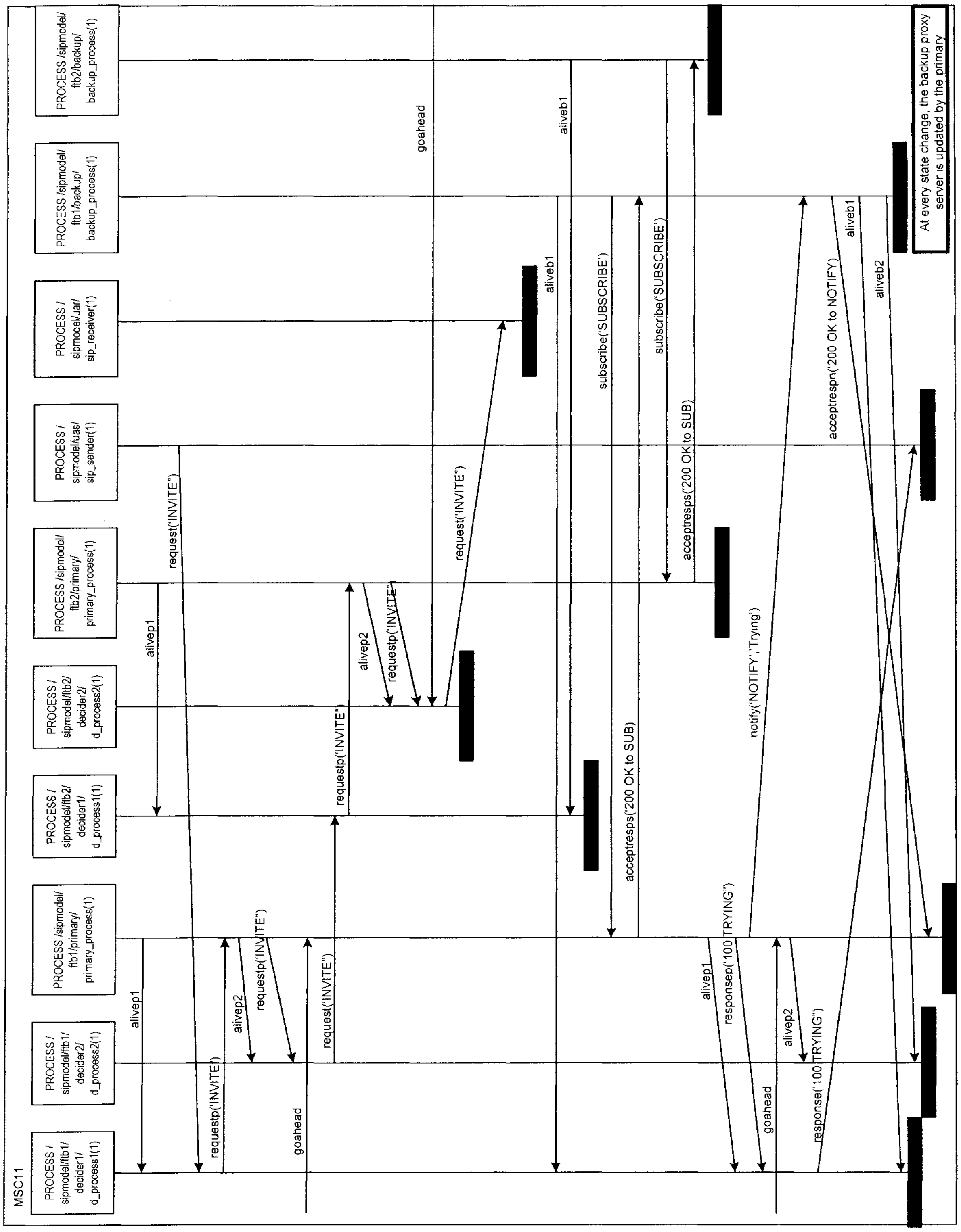

MSC11
PROCESS / sipmodel/ftb1/ decider1/ d_process1(1)
PROCESS / sipmodel/ftb1/ decider2/ d_process2(1)
PROCESS /sipmodel/ ftb1/primary/ primary_process(1)
PROCESS / sipmodel/ftb2/ decider1/ d_process1(1)
PROCESS / sipmodel/ftb2/ decider2/ d_process2(1)
PROCESS /sipmodel/ ftb2/primary/ primary_process(1)
PROCESS / sipmodel/uas/ sip_sender(1)
PROCESS / sipmodel/uar/ sip_receiver(1)
PROCESS /sipmodel/ ftb1/backup/ backup_process(1)
PROCESS /sipmodel/ ftb2/backup/ backup_process(1)
alivep1
request('INVITE')
requestp('INVITE')
alivep2
requestp('INVITE')
request('INVITE')
goahead
alivep1
requestp('INVITE')
alivep2
requestp('INVITE')
request('INVITE')
goahead
alivep1
alivep2
responsep('100 TRYING')
goahead
response('100 TRYING')
aliveb1
aliveb1
subscribe('SUBSCRIBE')
acceptresps('200 OK to SUB)
subscribe('SUBSCRIBE')
acceptresps('200 OK to SUB)
notify('NOTIFY','Trying')
acceptrespn('200 OK to NOTIFY')
aliveb1
aliveb2
At every state change, the backup proxy server is updated by the primary

Figure 5.11 MSC of Backup Proxy subscribing to the Primary after INVITE is sent

5.2 SDL Limitations

A few limitations have been imposed on our design due to the SDL tool. The first is that the time measurement within SDL cannot be a decimal value. They are limited to positive integer types since they are represented in SDL as time units rather than actual measured time. Within SIP however there are some timers which are in milliseconds and others in seconds. The timer value of 500 ms was thus made to 1 sec or 1 time unit, and then all of the other timers were modified according to the same scale.

The next limitation is that there is no mechanism within SDL which allows the intrusion of faults within the EFSM itself. The tool however maintains a relation between the system and the environment and thus several messages were added to the design in order to demonstrate the occurrence of faults. The first message is the *StopP* message which simulates a fault within the primary proxy server. This causes the server to terminate at the location where it is and then revert back to the idle state. This will assure that no *Alive* messages are sent to the Decider block and thus the backup proxy server will be used. The next message is the *StartP* message which is added in order to simulate the recovery of a primary proxy server. This message is only inputted at the idle state which assures that this message is received for recovery simulation and not erroneously from another component. If the primary proxy server is not in the idle state when this message is sent, it means that an error never occurred and thus, this message is disregarded.

As can be seen in the system design in Figure 4.1, both of these messages are fed from the environment. These messages are thus user controlled and although desirable cannot be sent dynamically within the tool itself. They are fed into the system externally as a means of displaying the fault occurrence and recovery within the SIP session.

The final limitation is that the SDL tool does not allow any outputs to be sent once a new state has been reached. The state is defined within SDL as a point wherein an input is being awaited. In our design, we need to send *Alive* messages while we are in any given state since there is no restriction as to when the proxy servers can issue these messages. Within SDL however we cannot send these messages after reaching a state so the alternative is to send the message before the state is reached. In this case however, if the primary proxy server fails at the new reached state then there will be inconsistencies within our design. The Decider block will receive an *Alive* message before the primary proxy server fails.

Furthermore, in the case of our design, there is a requirement to simulate faults however there must be a mechanism which illustrates the correct functionality of a SIP entity. For this reason, we included the *GoAhead* message which represents that no faults have occurred within the system and the SIP session establishment is in progress. Prior to reaching a state, we have created intermediate states wherein a decision must be made. If the *GoAhead* message is received, the session establishment continues without any faults. On the other hand if the *StopP* message is received, then a fault has occurred in the primary proxy server and the EFSM returns to the idle state. This decision is taken (i.e. the message is inputted from the environment) at every stage within the EFSM of the primary proxy server since a fault can occur anywhere within the SIP session establishment and termination phases.

5.3 Summary

This chapter summarized our simulation within the SDL tool. Furthermore, the MSC were compared to the theoretical simulations which were done in the previous chapter. It was shown that our Decider block handled the fault occurrence in all scenarios and that the dependency on timeout mechanisms was drastically reduced. Finally, we discuss the limitations of the SDL tool and the inclusion of messages fed from the environment as a means of overcoming these limitations.

Chapter 6 Conclusion and Future Work

6.1 Topic Overview

Several communication protocols exist for data transfer between end user agents however within the present market there is an increased need to send voice media over the Internet. This need has come about due to the merging of data and voice infrastructures as well as the addition of services within the domain of voice transfer. Several protocols have been introduced for this purpose including H.323 and SIP. These protocols allow connections between end users to be established and media to be sent between them. Furthermore, these protocols are interoperable with end users from different domains and are thus more flexible.

Although several alternatives exist, SIP has risen to be one of most widely used protocols for media transfers. The main reason is that SIP is more flexible with respect to the features which can be added. This allows SIP to be more customizable. Furthermore, SIP is a text based protocol which operates at the application layer making it simple and easy to use. In turn the simplicity of SIP makes it easier to develop and debug applications.

All of these advantages enhance the usage of SIP however there are some shortcomings which have to be examined. Although there is mention of security within SIP [4], there is scarce mention of the reliability of the entities within a SIP network. If end users are within the same domain then reliability is easily maintained however the transfer of media between end users in different domains is largely dependant on the availability of the resources between them. These resources include the links along which the media is sent and the proxy servers through which the media is transferred.

6.2 Contributions

In our thesis, we have identified the types of faults which may occur within a SIP session and effects of this on the overall system. The faults within a SIP session occur within the links or the proxy servers. The link faults are presently handled within SIP since the protocol contains a multi-layer model inclusive of a transport layer.

On the other hand, in the case of stateful proxies there are few mechanisms to handle the occurrence of faults. The protocol is equipped with timeout mechanisms which are used for fault detection while the server faults are handled by restarting the session (i.e. re-issuing the set of requests and responses). There are several drawbacks to this solution which include:

1) Prior to the occurrence of faults within a system, several resources are allocated in order to send requests and responses. Connections between end user agents are established and servers register with registrars within the domains to identify their availability. When a fault occurs, the message is re-issued through a new connection and thus a large amount of resources are wasted.
2) The timeout mechanism which is used for fault detection delays the overall recovery within the session. The occurrence of a fault within the system is not identified before the prescribed time elapses and thus recovery measures cannot be taken immediately. This delays the time taken for the session to be established.

3) The detection of faults within the system is conducted by the end user agents. The burden of retransmission is thus placed on the end user. Generally in communication protocols however, the end users should be unaware of the fault tolerance techniques available. They simply send the requests and receive the responses.

A thorough analysis of the disadvantages within the current SIP system was the motivation behind our work. We thus introduced a fault tolerance block which would detect the presence of faults immediately, eliminate the dependence on a timeout mechanism and remove the burden from the end user agents. Our first addition was the redundancy of the servers within the session. A primary backup scheme was used. This allows a backup server to be present to handle the current session in the presence of faults. The backup server is assumed to be non-faulty and can resume activity for the primary at any stage within the session.

We then introduce a Decider server which sends and receives requests and responses to the proxy server which is alive to handle the messages. If the primary server fails, the Decider server will not receive any *Alive* messages and thus will route the message through the backup server. This eliminates the dependence on the timeout present within SIP. Furthermore, the end UA can send requests and responses with the assurance that they will reach the intended destination. The method used for the message to reach the destination reliably is the role of the fault tolerance block which consists of a primary proxy server, a backup proxy server and a Decider server. Therefore the burden is removed from the end UA.

In order to complete our design, we identified the locations in which faults can occur within the session establishment and session termination phases. We determined the sequence of messages which should occur in order to assure that the session continues even with the presence of faults within the proxy servers of the system. Our proposed module was then simulated and verified in SDL. Faults were simulated within the system and message sequence charts were generated in order to verify the sequence of messages exchanged between the entities within the system. Two proxies were present within the system design in SDL so as to show the functionalities of the Decider blocks both with the end user agents as well as with the proxy servers. Finally the system was verified and all fault scenarios resulted as theoretically proposed.

Within our analysis of our proposed approach, we have compared our approach to other existent approaches with respect to the number of messages sent over the network as well as the time required for recovery from faults within a running SIP session. We have described that the time taken is reduced with the usage of a Decider server since the recovery process is completed before the initial INVITE request timer expires. The number of messages sent over the network increases however they are sent concurrently to other messages and therefore the time is not extended. Overall, the level of fault tolerance has been increased within our system so as to avoid inconsistencies or incorrectness within the data received which can be caused due to a time lag incurred by the restart process.

6.3 Future Work

Extending the research work presented in this thesis, involves the execution of performance analysis of the proposed Decider block. Although we have stated that the delay decreases while the number of messages increases, these factors have not been formally simulated within a tool such as NS-2 or OPNET. The factors to consider include network complexity, transmission delay, queuing delay, bandwidth usage, and packet loss. Analysis of these factors will prove the overall gain when using the Decider block within a SIP network. Furthermore, the performance analysis tools will allow a real-time simulation of our proposed approach indicating the actual time saved during the establishment and termination of a SIP session.

References

1. Goode, B. "Voice over Internet Protocol (VoIP)." In *Proc. of the IEEE* 90.9 (2002): 1495-1517.

2. Sinnreich, Henry. *Internet communications using SIP: delivering VoIP and multimedia services with Session Initiation Protocol.* New York: Wiley Computer Pub., 2001.

3. ITU-T Recommendation H.323. "Packet-Based Multimedia Communications Systems." February 1998.

4. Rosenberg, J., Schulzrinne, H., *et al.* "SIP: Session Initiation Protocol." IETF RFC 3261, June 2002.

5. Johnston, Alan B. *SIP: understanding the Session Initiation Protocol.* Boston: Artech House, 2001.

6. Camarillo, Gonzalo. *SIP Demystified.* New York: McGraw-Hill, 2002.

7. Dalgic, I., and H. Fang. "Comparison of H.323 and SIP for IP Telephony Signaling." In *Proc. of Photonics East* (1999).

8. Handley, M., and V. Jacobson. "SDP: Session Description Protocol." IETF RFC 2327, April 1998.

9. Schulzrinne, H., Casner, S., Frederick, R. and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications." IETF RFC 3550, July 2003.

10. Yennun, H., and C. Kintala. *Software Fault Tolerance*. 231-247. John Wiley & Sons, 1995.

11. Schneider, F.B. "Implementing Fault-tolerant services using the State Machine Approach: A tutorial." In *ACM Computing Survey* 22 (1990): 299-319.

12. Budhiraja, N., Marzullo K., Schneider F., and S. Toueg "Primary-Backup Protocols: Lower Bounds and Optimal Implementations." In *Proc. Third IFIP Conference on Dependable Computing for Critical Applications* (1992): 321-343.

13. Rosenberg, J., "Reliability of Provisional Responses in the Session Initiation Protocol (SIP)." IETF RFC 3262, June 2002.

14. Rosenberg, J., "The Session Initiation Protocol (SIP) UPDATE Method." IETF RFC 3311, September 2002.

15. Roach, A. B., "Session Initiation Protocol (SIP)-Specific Event Notification." IETF RFC 3265, June 2002.

16. Gulbrandsen, A., Vixie, P., and L. Esibov. "A DNS RR for specifying the location of services (DNS SRV)." IETF RFC 2782, February 2000.

17. Singh, K., and H. Schulzrinne. "Failover and Load Sharing in SIP Telephony." Tech. Rep. CUCS-011-04, Columbia University, Computer Science Department (2004).

18. Rosenberg, J., and H. Schulzrinne. "Session Initiation Protocol (SIP): Locating SIP Servers." IETF RFC 3263, June 2002.

19. Conrad, P., Jungmaier, A., Ross, C., Sim, W.-C., and M. T¨uxen. "Reliable IP Telephony Applications with SIP using RSerPool." In *Proc. of the SCI, Mobile/Wireless Computing and Communication Systems II* 10(2003).

20. Tuexen, M., Xie, Q., Stewart, R., Shore, M., Ong, L., Loughney, J., and M. Stillman. "Requirements for Reliable Server Pooling." IETF RFC 3237, January 2002.

21. Bozinovski, M., Gavrilovska, L., Prasad, R., and H.-P Schwefel. "Evaluation of a Fault-Tolerant Call Control System." In *Facta Universitatis Series: Electronics and Energetics* 17.1 (2004).

22. Doldi, Laurent. *Validation of Communication Systems with SDL*. England: Wiley & Sons, 2003.

23. Chan, K., and G.v. Bochmann. "Modeling IETF Session Initiation Protocol and its services in SDL." In *Proc. Eleventh SDL Forum* 2708 (2003): 352-373.