

Verification of the SIP Transaction Using Coloured Petri Nets

Lin Liu

School of Computer and Information Science
University of South Australia
Mawson Lakes, SA 5095, Australia
Email: lin.liu@unisa.edu.au

Abstract

The Session Initiation Protocol (SIP) is one of the leading protocols for multimedia control over the Internet, including initiating, maintaining and terminating multimedia sessions. The protocol uses transactions to complete the control tasks. In this paper we focus on the INVITE transaction of SIP, which is used to initiate a session. SIP is designed to operate over a transport protocol that can be reliable or unreliable. Our previous work has verified the functional properties of the INVITE transaction over a reliable transport medium, using Coloured Petri Nets (CPNs). In this paper, we use CPNs to model and analyse SIP INVITE transaction when the medium is unreliable. The verification reveals similar problem as that in the case of a reliable medium, i.e. the transaction may terminate in an undesirable state while one communication party is still waiting for a response from its peer. Additionally with an unreliable medium, the transaction has undesirable terminal states in which retransmitted requests may lead to erroneous operation. This result provides theoretical evidence and timely support for the Internet Draft that has been recently submitted to the Internet Engineering Task Force to propose updates to SIP INVITE transaction.

Keywords: Session Initiation Protocol, Coloured Petri Nets, protocol verification

1 Introduction

In (Ding & Liu 2008), we have investigated the properties of the Session Initiation Protocol (SIP) (Rosenberg et al. 2002), assuming that the transport medium over which the protocol operates is reliable. This paper continues the investigation by analysing the properties of SIP when considering an unreliable medium which may reorder and/or lose messages.

SIP has been widely used for establishing, maintaining and tearing down multimedia sessions over the Internet, and it has become increasingly popular in Voice over IP applications. SIP is developed by the Internet Engineering Task Force (IETF) and is specified in Request for Comments (RFC) 3261 (Rosenberg et al. 2002). Currently work is still being carried out within IETF, to maintain and continue the development of this protocol. Modelling and analysing SIP specification using formal methods can help in assuring that the contents of RFC 3261 are correct, unambiguous, and easy to understand. A well-defined and

verified protocol specification will reduce the cost for its implementation and maintenance, therefore from the point of view of protocol engineering, verification is also an important step of the life-cycle of protocol development (Holzmann 1991).

We use Coloured Petri Nets (CPNs) (Jensen 1997) as the modelling and analysing technique, due to many successful applications of CPNs in verifying communication protocols (Billington et al. 2004, Kristensen et al. 2004). CPNs also have their well-developed supporting software package, the CPN Tools (Homepage of the CPN Tools).

As mentioned in (Ding & Liu 2008), to our best knowledge, most of the publications related to SIP are in the areas of interworking of SIP and another multimedia communication protocol suit, H.323, and SIP services. Only a handful papers have been found on modelling and/or analysing SIP. Gehlot & Hayrapetyan (2006) have modelled a SIP-based discovery protocol for the Multi-Channel Service Oriented Architecture, which uses the non-INVITE transaction as one of the basic components for web services in a mobile environment. However, no analysis results have been reported on this SIP-based discovery protocol. In (Wan et al. 2007), the authors have modelled SIP with the purpose of analysing SIP security mechanism. They have analysed the state space of the CPN model in a typical attack scenario. Other related work we have found include (Peng et al. 2007) and (Sun et al. 2007). The former creates a Petri net model for a specific SIP call flow and shows that it is deadlock free. However, no modelling or analysis work is done on SIP's operation in general. The latter presents an extended finite state machine model for SIP, but no analysis results are reported.

SIP carries out tasks through a series of transactions. The two main SIP transactions are the INVITE transaction for setting up a session, and the non-INVITE transaction for maintaining and closing down a session. Our current work is aimed at verifying the functional properties of the INVITE transaction. According to (Rosenberg et al. 2002), SIP INVITE transaction can operate over a reliable (e.g. TCP) or an unreliable (e.g. UDP) transport medium. To verify the INVITE transaction, we take an incremental approach. We firstly analyse the transaction's behaviour over a perfect (i.e. reliable) medium, then its operation over an imperfect one, because a lossy and/or reordering medium may mask some problems that will only be detected with a perfect medium. In (Ding & Liu 2008), we assume a reliable transport medium is used. A deadlock has been detected, a revision to the specification of SIP INVITE transaction has been proposed, and the revised specification is shown to be deadlock free. In this paper, we move on to the more general case, where the medium may lose and/or reorder messages. This brings up new modelling and analysis challenges that we do not have in

the case when assuming a reliable medium.

The rest of the paper is organised as follows. Section 2 introduces SIP INVITE transaction. Section 3 presents the CPN for the INVITE transaction over an unreliable medium. State space analysis of the transaction is then described in Section 4. Finally, Section 5 concludes the work and suggests future research.

2 The INVITE Transaction of SIP

SIP is structured into four layers, each of which carries out a set of functions. From bottom to top of the structure are the syntax and encoding, transport, transaction, and transaction user (TU) layers. The syntax and encoding layer specifies the format and structure of a SIP message. The transport layer transmits/receives SIP messages to/from the underlying transport medium. On top of SIP transport layer is the transaction layer, with each transaction consisting of a client transaction sending requests and a server transaction responding to requests. Residing in the top layer are the TUs, which creates and destroys SIP transactions, and utilises services provided by the transaction layer.

The most important layer of SIP is the transaction layer. It is responsible for matching request and response messages, retransmitting messages when the transport medium is unreliable, and handling time-outs. A SIP message is a request from a client to a server, or a response from a server to a client. Each request message carries a method (such as INVITE or ACK) to invoke a particular operation on a server. Each response has a status code indicating the acceptance, rejection or redirection of a SIP request (Table 1). The first message transmitted during a SIP transaction is always a request message, and based on the type of the request, the transaction is known as an INVITE transaction (if it is an INVITE request) or a non-INVITE transaction (if it is a request other than INVITE or ACK). An INVITE transaction is used to establish a session, while a non-INVITE transaction is used to maintain and tear down a session.

The INVITE client and server transactions are defined in RFC 3261 using two state machines, as shown in Figure 1. The operations that are carried out only when the underlying medium is unreliable are highlighted in the figure with bold-faced characters.

When TU at the client side wants to initiate a session, it creates an INVITE client transaction, and passes an INVITE request to the transaction. An INVITE client transaction (Figure 1(a)) has four states: *Calling*, *Proceeding*, *Completed*, and *Terminated*. When the transaction is created, it enters the *Calling* state, passes the INVITE request received from TU to SIP transport layer for transmitting to the server side, and starts *Timer B*. *Timer A* is started only when the transport medium is unreliable.

When the transaction is in its *Calling* state, one of the following six events can occur.

- *Timer A* fires. The transaction resets the timer and retransmits the INVITE request;
- *Timer B* fires. The transaction enters its *Terminated* state;

- An transport error is reported by SIP transport layer when it tried to send an INVITE request over the network. The transaction informs its TU of the error and enters the *Terminated* state;
- A provisional response *1xx* (Table 1) is received. The transaction passes the response to its TU and enters the *Proceeding* state, waiting for further responses;
- A final success response *2xx* is received, indicating the server has accepted the INVITE request. The transaction informs its TU of the response and enters its *Terminated* state;
- A final non-success response (*300-699*) is received, indicating the server received but did not accept the INVITE request. The transaction passes the response to its TU, creates an ACK request and passes it to SIP transport layer, then it enters the *Completed* state.

When *Proceeding*, the client transaction may:

- receive any number of provisional responses *1xx*, and stays in the *Proceeding* state;
- receive a final success response, and enters the *Terminated* state;
- receive a final non-success response, and moves to the *Completed* state after creating and sending an *ACK*.

The purpose of the *Completed* state is to absorb *300-699* responses retransmitted by server when the medium is unreliable. When this state is entered,

- *Timer D* is started. When the timer expires, the transaction enters its *Terminated* state;
- if a *300-699* response is received before *Timer D* expires, the transaction creates and sends an *ACK* and stays in the same state;
- if a transport error occurs when SIP transport layer is sending an *ACK*, the TU is informed and the transaction is *Terminated*.

When the client transaction enters its *Terminated* state, it is destroyed by the TU immediately.

At the server side, when the TU receives an INVITE request, it creates a server transaction (Figure 1(b)). The transaction sends a *100* (trying) response if the TU does not generate a response within 200 milliseconds (ms). The transaction then enters the *Proceeding* state. The other three states of the transaction are: *Completed*, *Confirmed* and *Terminated*.

While *Proceeding*, the server transaction may:

- pass any provisional responses *101-199* generated by its TU to SIP transport layer, and stays in the same state;
- receive an INVITE request retransmitted by the client transaction, then the server transaction retransmits the provisional response it previously received from its TU and stays *Proceeding*;
- get a final non-success response *300-699* from its TU, then it moves to the *Completed* state after sending the response;
- receive a transport error report from SIP transport layer, the transaction then enters the *Terminated* state;
- get a final success response *2xx* from its TU, the transaction is then *Terminated* after sending the response.

Once the *Completed* state is entered, *Timer H* is started. When the transport is unreliable, *Timer G* is also started to control the time for each retransmission of the *300-699* response it previously received from its TU while in the *Proceeding* state. Then one of the following five events can occur.

Table 1: SIP response messages

Response	Function
<i>1xx</i>	provisional, request received but not yet accepted
<i>2xx</i>	success, request received and accepted
<i>3xx</i>	redirection, further action required
<i>4xx</i>	bad syntax in the request
<i>5xx</i>	server failed to answer the request
<i>6xx</i>	no server can answer the request
note: <i>xx</i> stands for numbers from 00 to 99	

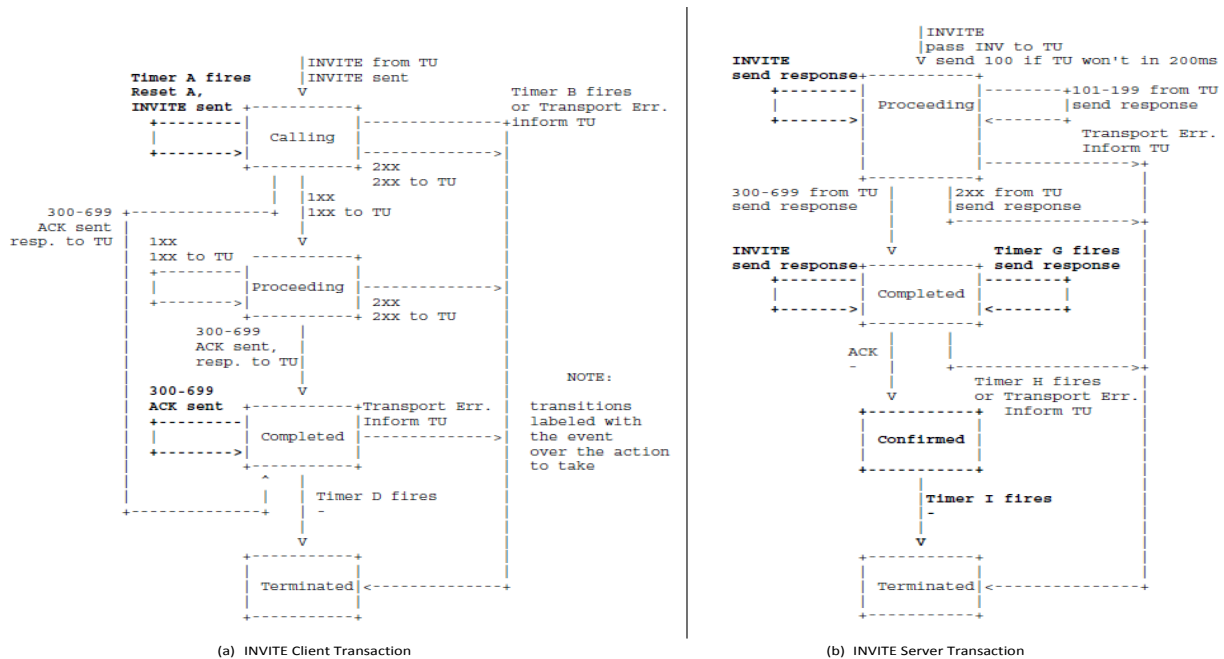


Figure 1: State Machines defining SIP INVITE transaction (Rosenberg et al. 2002)

- A retransmitted INVITE request is received. The transaction retransmits the *300-699* response it received from its TU while it was *Proceeding* and stays in the same state;
- *Timer G* fires. The transaction also retransmits the *300-699* response, resets the timer and stays in the same state;
- *Timer H* fires. The transaction moves to its *Terminated* state;
- A transport error occurs when SIP transport layer sends a response. The transaction is then *Terminated*;
- An ACK is received from the client side. The transaction does nothing except for changing its state to *Confirmed*.

When the *Confirmed* state is entered, *Timer I* is started, the transaction is waiting for ACKs triggered by the retransmissions of *300-699* responses. When *Timer I* fires, the *Terminated* state is entered, and the server transaction is destroyed by its TU.

3 Modelling the INVITE Transaction

By the looks of it, modelling state machines using CPNs could be straightforward, as we can model states of an entity using CPN places, and actions that cause state changes using CPN transitions. While this may be a general rule to follow when modelling state machines, for any particular system, the modelling process is not a straight translation from state machines to CPNs. With the INVITE transaction of SIP, due to the incompleteness of the state machines and the inconsistency between the state machine and the plain text description given in RFC 3261, assumptions must be made before modelling. Meanwhile, details that are not shown on the state machines but are related to the operations of the transaction, i.e. relations between different timers, need to be sorted out as well. In the following we firstly describe these aspects, then we introduce the CPN model for the INVITE transaction over an unreliable medium.

3.1 Assumptions

According to the plain text description provided in Section 17.1.1 of (Rosenberg et al. 2002), the IN-

VITE client transaction must firstly enter its *Calling* state before receiving an INVITE request from its TU. However, the state machine (Figure 1(a)) shows that the client transaction receives an INVITE from TU and sends it to SIP transport layer before entering the *Calling* state, i.e. before the transaction is created, which is impossible. So as in (Ding & Liu 2008), here we assume that only when the client transaction has been created and is *Calling*, the transaction can receive an INVITE from TU.

With the server transaction, as in (Ding & Liu 2008), we assume that the transaction does not know that the TU will generate a response within 200 ms after it is created by TU, i.e. the server transaction must generate and send a *100 Trying* response after it is created. As the server transaction has to be in a state when receiving an INVITE request and sending the *100 Trying* response, we denote a new state for the server transaction, *ProceedingT*. The transaction enters this state immediately after it is created and receives an INVITE request from its TU. The only event can occur in this state is a *100 Trying* response is sent by the server transaction.

While the above two assumptions are the same as in (Ding & Liu 2008), the following three are new for the case when the underlying medium may lose or reorder messages.

Figure 1(a) shows that when the client transaction is in the *Completed* state, it can receive only *300-699* responses. However, this is not true when the medium is unreliable. From Figure 1(b) the server transaction can send *300-699* responses after sending *1xx* responses. However an unreliable medium may reorder messages, *1xx* responses may arrive at the client side after *300-699* responses. So it is possible for the client transaction to receive a *1xx* response when it is in the *Completed* state. We assume that in this case the client transaction discards the provisional response and stays in the *Completed* state.

When the server transaction is in its *Completed* state, *Timer G* can fire (Figure 1(b)). The state machine does not specify the event of resetting the timer. However, based on the text description of RFC 3261, *Timer G* can fire more than once, so we assume that *Timer G* is reset every time when it fires.

The server state machine specifies that, when the

server transaction is in its *Confirmed* state, the only event that can occur is the firing of *Timer I*. However, it is still possible for the server side to receive retransmitted INVITEs or ACKs when the transport medium is unreliable. So we assume that when the server transaction is *Confirmed*, when an INVITE or an ACK is received, the transaction stays in the same state and the INVITE or ACK is simply discarded.

3.2 Relations between Timers

Referring to Figure 1(a), the client transaction uses three timers: *A*, *B* and *D*. *Timer B* sets up the maximum time that the client transaction would wait in its *Calling* state for a provisional or final response from the server side. This timer is used no matter over what transport medium the transaction is running. *Timer A* is used only when the medium is unreliable, to control retransmissions of INVITE requests. So it is not considered in (Ding & Liu 2008), but it must be included in our model for unreliable transport medium. *Timer D* also only plays its role when the medium is unreliable because its value is set to zero for reliable transport and 32 seconds for unreliable transport.

Among the three timers, *A* and *B* are related. However, the way in which they are related is not explicitly described in RFC 3261. To capture all possible occurrence sequences of events using CPNs, we need to firstly make clear the relationship between the two timers. From the RFC, the initial value of *Timer A* is $T1$, whose default value is 500ms, equal to the estimated round trip time between the client and server. Every time when *Timer A* fires, it is reset to the value that doubles its previous value. The value of *Timer B* is $64 * T1$, so we can see that before *Timer B* fires, *Timer A* can occur 6 times (less than 6 times if a response is received or a transport error occurs before *Timer B* fires), at the intervals $T1$, $2 * T1$, $4 * T1$, $8 * T1$, $16 * T1$, $32 * T1$ respectively (note that sum of the intervals is $63 * T1$, just less than $64 * T1$, this is why *Timer A* can fire at most 6 times before *Timer B* expires). Every time when *Timer A* fires, the INVITE request is retransmitted. Because when the client transaction is created, the original INVITE request is transmitted, at most 7 INVITE requests can be sent by the client transaction before *Timer B* fires.

The server transaction also has three timers: *G*, *H* and *I* (Figure 1(b)). *Timer H* is used for both reliable and unreliable medium, to set up the maximum time that the server transaction would wait in its *Completed* state before an ACK is received. *Timer G* is only used for unreliable transport, to control retransmissions of *300-699* responses. When the *Completed* state is entered, *Timer H* is set to $64 * T1$, and *Timer G* is set to $T1$. When *Timer G* fires for the i th time ($i \geq 2$), it is reset to the minimum of $2^{i-1} * T1$ and $T2$ ($T2 = 8 * T1$). Therefore, before *Timer H* expires, *Timer G* can occur up to 10 times, at the intervals $T1$, $2 * T1$, $4 * T1$, $8 * T1$, $8 * T1$, $8 * T1$, $8 * T1$, $8 * T1$, $8 * T1$, $8 * T1$ respectively. If an ACK is received or a transport error occurs before *Timer H* fires, *Timer G* occurs less than 10 times. *Timer I* is set to fire in zero seconds for a reliable transport medium, and 5 seconds for an unreliable medium.

3.3 CPN Model of the INVITE Transaction over an Unreliable Medium

Figures 2 (the CPN) and 3 (declarations) show the CPN model for the INVITE transaction over an unreliable medium. The model is based on the state machines in Figure 1. It extends and revises the CPN model presented in (Ding & Liu 2008), by considering the additional operations of the transaction for unreliable medium (i.e. the highlighted parts of

Figure 1), and by modelling an unreliable transport medium. Same naming conventions are used here as in (Ding & Liu 2008). To distinguish a server transaction's state from a client transaction's state with the same name, a capitalised *S* is appended to the name of the state of the server transaction (except for the *proceedingT* state). For example, *proceedingS* represents the *Proceeding* state of the server transaction while *proceeding* represents the *Proceeding* state of the client transaction. SIP response messages (Table 1) are named as follows: *r100* represents a *100 Trying* response; *r101* is for a provisional response with a status code between 101 and 199; *r2xx* for a *2xx* response; and *r3xx* for a *300-699* response.

3.3.1 Client Transaction

The left part of the CPN model (Figure 2), including places *Client* and *INVITE Sent*, and the transitions connected to them, models the client transaction state machine (Figure 1(a)).

CPN place *Client* is typed with colour set *STATEC* (lines 1 and 2 of Figure 3), modelling the states of the INVITE client transaction. The initial marking of *Client* is *calling*. Place *INVITE Sent* is typed by colour set *INT* (line 8 of Figure 3), used to count the number of INVITE requests that have been transmitted and retransmitted.

Five transitions are connected to place *Client*. *Send Request* models how the transaction passes the original INVITE request to SIP transport layer for transmission (i.e. putting an INVITE token in place *Requests*). It is enabled only if the *Client* is *calling* and *INVITE Sent* contains an integer 0 (no INVITE request has been sent).

Receive Response models how the client transaction receives responses and sends ACKs. It is enabled when a response is received and the *Client* is not *terminated*. If the client transaction receives a *300-699* response (*r3xx*), an ACK is passed to SIP transport layer (see the arc from *Receive Response* to place *Requests*), and the *Client* changes to *completed*. Refer to the inscription of the arc from *Receive Response* to place *Client*, if the received response is *r100*, *r101* or *r2xx*, no ACK is sent; when the response is *r100* or *r101*, the *Client* changes to *proceeding*; and when the response is *r2xx*, the *Client* changes to *terminated*. This arc inscription also models the assumption made in Section 3.1 about receiving a provisional response by the client transaction in its *Completed* state.

We could add a transition to the CPN in (Ding & Liu 2008) to model *Timer A*. However, given the relation between *Timer A* and *Timer B* (Section 3.2), it is possible to model the two timers using one transition *Timer A* or *B* (Figure 2). To model that *Timer A* cannot be started before an INVITE request is sent, we let *INVITE Sent* be an input place of *Timer A* or *B*, and the transition is enabled only when an integer greater than or equal to 1 is in *INVITE Sent* (see the guard $[a \geq 1]$) and the *Client* is *calling*. The initial marking of *INVITE Sent* is 0. When *Send Request* or *Timer A* or *B* occurs (an INVITE request is sent), the integer value is incremented by 1. Based on the discussion in Section 3.2, *Timer B* does not expire until *Timer A* has expired 6 times, or 7 INVITE requests have been sent. So the inscription of the arc from *INVITE Sent* to *Timer A* or *B* checks the value of variable *a*. When $a < 7$, only *Timer A* can fire, so the occurrence of *Timer A* or *B* does not change the marking of *Client*, but puts an INVITE in place *Requests*. When $a = 7$, the occurrence of the transition models the firing of *Timer B*, which changes the marking of *Client* from *calling* to *terminated*, but does not put anything in place *Requests*.

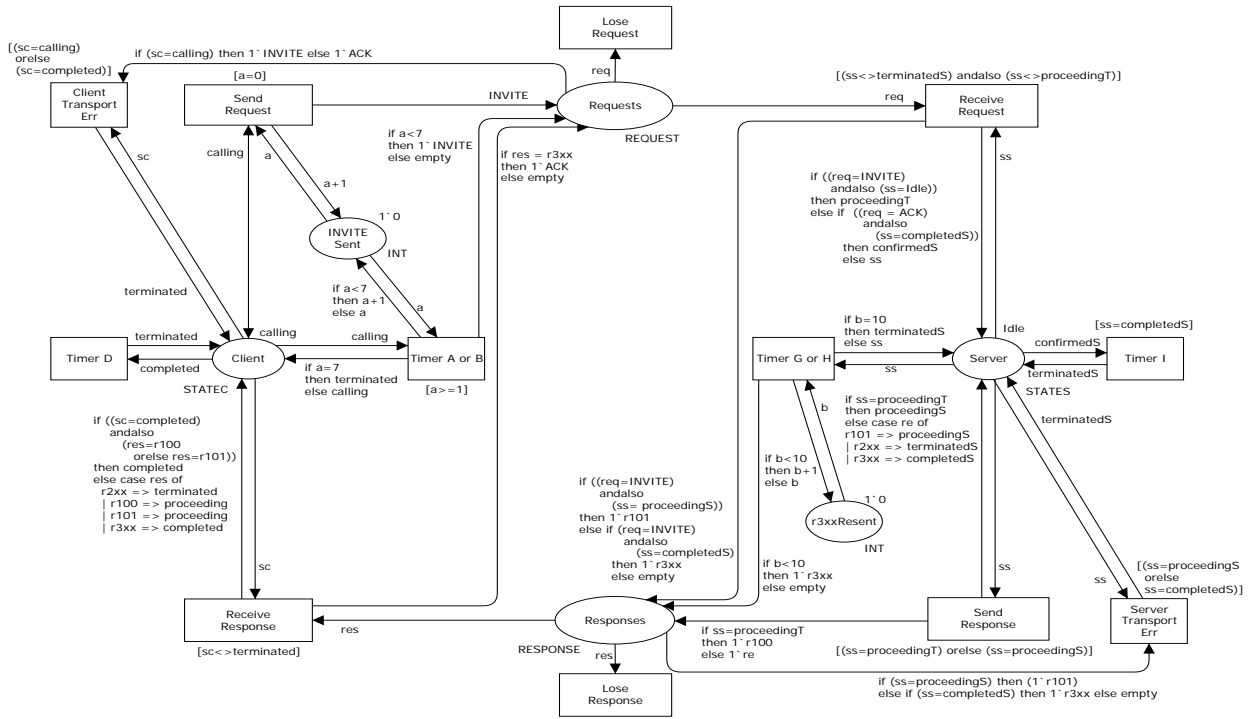


Figure 2: CPN model of the INVITE transaction

```

(*---states of client transaction---*)
1 colset STATEC = with calling | proceeding | completed
2 | terminated;
(*---states of server transaction---*)
3 colset STATES = with Idle | proceedingT | proceedingS
4 | confirmedS | completedS | terminatedS;
(*---request messages sent by client---*)
5 colset REQUEST = with INVITE | ACK;
(*---response messages sent by server---*)
6 colset RESPONSE = with r100 | r101 | r2xx | r3xx;
7 colset Response = subset RESPONSE with [r101, r2xx, r3xx];
(*---Integer for counting transmitted messages---*)
8 colset INT = int with 0..11;
(*---variables---*)
9 var sc : STATEC;
10 var ss : STATES;
11 var req : REQUEST;
12 var re : Response;
13 var res : RESPONSE;
14 var a,b : INT;

```

Figure 3: Declarations of the CPN model

Transition **Timer D** is enabled once a **completed** is in **Client**, and its occurrence changes **Client** to **terminated**, modelling the firing of **Timer D**.

Transition **Client Transport Err** is enabled when the **Client** is **completed**. Its occurrence also changes **Client** to **terminated**. When an error is reported by SIP transport layer, the ACK that has been passed to it from the transaction layer will not be sent to the server side, so when **Client Transport Err** occurs, the ACK that has been put in place **Requests** is destroyed (see the inscription of the arc from **Client Transport Err** to **Requests**). From Figure 1(a), a transport error can occur when the client transaction is *Calling*, so **Client Transport Err** is enabled as well when the **Client** is *calling* (see the guard of **Client Transport Err**).

3.3.2 Server Transaction

The right part of Figure 2, i.e. places **Server** and **r3xxResent**, and the five transitions connected to them model the INVITE server transaction defined in Figure 1(b).

Place **Server** is typed by colour set **STATES** (lines

3 and 4 of Figure 3), modelling the states of the server transaction. **ProceedingT** models the new state *ProceedingT* that we add to the server transaction (Section 3.1). Because the transaction is created and enters the *ProceedingT* only *after* the TU has received an INVITE request from the client side via SIP transport layer directly, the initial marking for place **Server** cannot be **proceedingT**. However, a place has to have an initial marking, so we let the place's initial marking be **Idle** (line 3 of Figure 3), modelling that the server side is ready to receive an INVITE request from the client side. Place **r3xxResent** is typed by **INT** and it records the number of **r3xx** retransmitted when **Timer G** fires.

Transition **Receive Request** models the reception of an INVITE or ACK request. When the transition occurs upon receiving an INVITE request and the **Server** is **Idle**, a **proceedingT** is created in the **Server** place. In this case and only in this case, transition **Receive Request** models the operation of the TU instead of the server transaction of receiving an INVITE request from the client side.

Once the **Server** is **proceedingT**, transition **Send Response** is enabled, thus a **r100** can be put into **Responses** (see the inscription of the arc from transition **Send Response** to place **Responses**), and the state of **Server** is changed to **proceedingS** (see the then clause of the inscription of the arc from **Send Response** to **Server**). In the **proceedingS** state, **Send Response** is again enabled. When it occurs, a **r101**, **r2xx** or **r3xx** response is put into place **Responses**. This is represented by the variable **re** included in the else clause of the inscription of arc from **Send Response** to **Responses** where **re** is of type **Response** (line 7 of Figure 3). Meanwhile, a **proceedingS** (if a **r101** is put in **Responses**), **completedS** (for **r3xx**) or **terminatedS** (for **r2xx**) is put in the **Server** place (see the else clause of the inscriptions of the outgoing arcs of **Send Response**).

While the **Server** is **completedS**, if an ACK is received, the occurrence of **Receive Request** changes the **Server** to **confirmedS**. We have assumed that the server transaction can receive INVITE or ACK

requests when it is `confirmedS` (Section 3.1), the last `else` clause of the inscription of the arc from `Receive Request` to `Server` models the server transaction stays in the same state and do not send any response. The guard of `Receive Request` models that the server transaction cannot receive any requests after it is *Terminated* because it is destroyed by TU after the *Terminated* state is entered.

From Figure 1(b) if the medium is unreliable, when the `Server` is `proceedingS` or `completedS`, a response (`r101` or `r3xx`) is sent upon receiving an `INVITE` retransmitted by the client. To model this, we add to the CPN model in (Ding & Liu 2008) an arc from `Receive Request` to `Responses` (see Figure 2).

Similar to the modelling of *Timer A* and *Timer B* for the client transaction, *Timer G* and *Timer H* are modelled by a single transition `Timer G` or `H`. From Section 3.2, *Timer H* does not fire until *Timer G* has fired for 10 times, or 10 `r3xx` responses have been sent. We let `r3xxResent` be an input place of `Timer G` or `H`, so the number of `r3xx` responses sent can be known via checking the value of variable `b`. When `b < 10`, only *Timer G* fires, so the occurrence of transition `Timer G` or `H` only puts a `r3xx` into `Responses` without changing the `Server`'s state. When `b = 10`, the occurrence of this transition models the firing of *Timer H*, which changes the `Server` from `completedS` to `terminatedS`, and does not put any response into place `Responses`.

Transition `Server Transport Err` models transport errors occurring at the server side. It is enabled when the `Server` is `proceedingS` or `completedS` (i.e. after a response is passed to SIP transport layer). When it occurs, the response that has just been put into `Responses` is removed (see the inscription of the arcs from `Server Transport Err` to `Responses`).

Timer I is modelled using transition `Timer I`, which is enabled when the `server` is `completedS` and its occurrence changes the `Server` to `terminatedS`.

3.3.3 Transport Medium

The middle part of the CPN model (places `Requests` and `Responses`, and transitions `Lose Request` and `Lose Response`) models SIP transport layer and the underlying transport medium. `Requests` models the transmission of requests from the client side to the server side; whereas `Responses` models the transmission of responses in the reverse direction. In (Ding & Liu 2008), the two places are typed by CPN lists (a first in first out queue), as the transport medium does not reorder or lose any messages. When the transport medium is unreliable, we cannot use lists. Instead we let the colour sets of `Requests` and `Responses` be multisets of possible requests and responses respectively (lines 5 and 6 of Figure 3), so that an occurrence of an output transition of `Requests` or `Responses` destroys a randomly picked request or response. This models that the transport medium may reorder messages, i.e. messages are not received in the order they are sent (put into the `Requests` or `Responses` place). To model message loss, we use two transitions `Lose Request` and `Lose Response`, whose occurrences destroy requests and responses from places `Requests` and `Responses` respectively.

4 Analysing the INVITE Transaction

4.1 Desirable Properties

A protocol can fail if it has deadlocks or livelocks (Holzmann 1991). So we firstly check if the `INVITE` transaction is *absence of deadlocks* and *absence of livelocks*. A deadlock is an undesired terminal state of a system. It appears as an undesired dead marking in the state space of the CPN model of the system.

A marking of a CPN is dead if no transitions are enabled in it (Jensen 1997). For the `INVITE` transaction, a desirable terminal state must have both the client and the server transactions in their *Terminated* state (see Figure 1), and ideally no messages are left in the communication channels, i.e. places `Requests` and `Responses` are empty. A livelock is a cycle of the state space that once entered, can never be left, and within which no progress is made with respect to the purpose of the system. We also expect that the `INVITE` transaction has *no dead code*, action that is specified but never executed. A dead code is shown as a dead transition (a transition that is not enabled in any marking) of the CPN model of a system.

To verify the three properties of the `INVITE` transaction, we use the state space analysis method of CPNs (Jensen et al. 2007) with the support of the CPN Tools (Homepage of the CPN Tools).

4.2 The CPN Model for Analysis

As we can see from the state machine (Figure 1(b)), unlimited number of `101-199` responses can be generated by the TU at the server side. Correspondingly, with the CPN model (Figure 2), transition `Send Responses` can occur unlimited number of times when the `Server` is `proceedingS`, so an arbitrarily large number of `r101s` can be put in place `Responses`, thus the state space of the model is infinite. To use the state space analysis method, we must limit the capacity of place `Responses` and the number of `r101` put into it. This is what we did in (Ding & Liu 2008).

However, with unreliable medium, limiting the capacity of `Responses` is not adequate. From Section 3.2, up to 7 `INVITE` requests can be put in `Requests`. Every time when a `r3xx` is received by the client transaction, an `ACK` is put into `Requests`. One `r3xx` can be sent by the server transaction when it is in the *Proceeding* state. Also every time when the transaction receives a retransmitted `INVITE` in the *Completed* state, or when *Timer G* fires, a `r3xx` is sent. As there are up to 6 retransmitted `INVITE` requests, 10 firings of *Timer G*, 17 `r3xx` can be sent. So at the client side up to 17 `ACKs` can be put in place `Requests`. This means place `Requests` can have up to 24 requests, including 7 `INVITEs` and 17 `ACKs`. Given this number and the possible combinations of markings of places `Requests` and `Responses`, even we limit the capacity of `Responses`, we still have a state space that is too large to be handled by the CPN Tools. So we must limit the capacity of place `Requests` and/or the number of `r3xx` sent (thus the number of `ACKs` put into place `Requests`).

The general idea of setting up the restrictions is: on one hand to avoid state space explosion, on the other hand to not to lose generality so that the restricted model captures most scenarios in practice. Based on this, we decide to set the capacities of both places to be 3. The reason for it is discussed below.

From Section 3.2, `INVITE` requests are transmitted at intervals $0, T1, 2^*T1, 4^*T1, 8^*T1, 16^*T1,$ and 32^*T1 . So the minimum time difference between sending the i th ($0 \leq i \leq 4$) and the $(i+2)$ th `INVITE` requests is 3^*T1 , which happens when $i=0$ (the original `INVITE` request is sent) followed by the expirations of *Timer A* for the first and second time. In practice, it is seldom that after 3^*T1 (estimated time for 3 round trips) or longer, a message is still not received by the server, i.e. it is very rare to have three `INVITE` requests exist in the channel (place `Requests`) at the same time. So setting the capacity of `Requests` to 3 will allow us to model the rare situation, thus cover most scenarios in practice. Furthermore, an `ACK` also can be put into place `Requests`, so our restricted model should allow `INVITE` and `ACK`

requests to exist in place **Requests** at the same time thus we can capture the message reordering behaviour of the medium. We know that an ACK is sent only after a **r3xx** response is received, and a **r3xx** is sent by the server only after the receipt of the original INVITE request. So after a **r3xx** is received at the client side, the client is no longer in its *Calling* state and the first INVITE must have arrived at the server side (removed from place **Requests**). So the ACK can be put into the place when its capacity is 3, thus this place can have both INVITE and ACK at the same time.

Four different types of responses can be put in **Responses**: **r100**, **r101**, **r2xx** and **r3xx**. **r100** and **r2xx** are sent only once respectively, and **r101** and **r3xx** can be retransmitted. Since it is not specified in RFC 3261 how many **r101** provisional responses can be generated by the TU at the server side, an arbitrarily large number of **r101** can be put into **Responses**. So we firstly limit the number of **r101** sent by the server side. To not lose generality, we restrict the maximum number of **r101** to 3 so that we can have multiple **r101**s in the system. This restriction is implemented by adding an input place (typed by INT) to transition **Send Responses**, to count the number of **r101**s in place **Responses**, and by adding a guard to the transition so that its occurrence does not put **r101** in **Responses** when the number stored in the counter place is 3. Furthermore, given that once a final non-success response **r3xx** is generated by TU, a final success response **r2xx** is not generated, and vice versa, setting the capacity of **Responses** to 3 will allow the two types of provisional responses (**r100** and **r101**) and either **r3xx** or a **r2xx** response to exist in place **Responses** at the same time. So we have the opportunity to investigate the consequences of re-ordering of these responses by the transport medium.

To model a reliable transport medium we can type each of the places **Requests** and **Responses** by a CPN list. Then we can easily set up the capacity of each place by using a CPN guard (to restrict the number of occurrences of transitions that deposit tokens into the place (Ding & Liu 2008)). To model an unreliable medium, we have to use multisets to type the places (Figure 2). Unfortunately CPNs do not provide a straightforward approach to check the number of tokens in a place typed by a multiset. So to restrict the capacities of places **Requests** and **Responses** we use the idea similar to the anti-place modelling technique (Homepage of the CPN Tools).

Take place **Requests** as an example (see Figure 4 where to save space transition **Lose Request** is not shown). To limit its capacity to 3, we create an anti-place **capReq** with type INT and initial marking 0. Every time when the occurrence of a transition deposits a message in **Requests**, the value stored in **capReq** is incremented by 1, and every time when the occurrence of a transition removes a message from **Requests**, the value stored in **capReq** is decreased by 1. We also add to every transition whose occurrence deposits a message in place **Requests** a guard **cap<3**, so that once the maximum value stored in **capReq** is 3 (correspondingly 3 messages are in **Requests**), no more message can be put into **Requests**. In this way, the capacity of place **Requests** is limited to 3. Note that transition **Client Transport Error** is also connected to place **Requests** and its occurrence removes a message from this place. However, once it occurs, the **Client** becomes **terminated**, so none of the transitions connected to **Requests** can occur, and the number of messages in **Requests** cannot be changed thereafter. Thus we do not consider the impact of this transition in terms of the capacity of **Requests**.

We call the CPN model with the restrictions on capacities of places **Requests** and **Responses** and the number of **r101** sent, the *Restricted CPN model*.

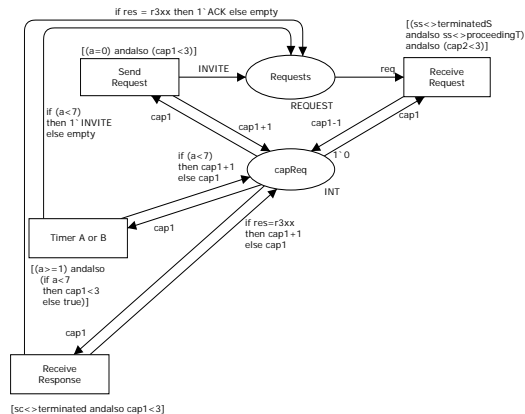


Figure 4: Illustration of the idea of “anti-place”

In the following an incremental approach is used to analyse the model. Firstly we assume that the medium may reorder messages only and no messages are lost. This corresponds to analysing the restricted CPN model without transitions **Lose Request** and **Lose Response**. Then we consider the case when the medium may reorder *and* lose message, i.e. the restricted model. The reason for taking this approach is similar to why we decided to firstly investigate the transaction over a reliable medium, i.e. behaviour of the medium (instead of the transaction itself) may mask problems of the transaction itself.

4.3 Reordering Medium

The state space report (Table 2) generated by the CPN Tools shows that the full state space has 30982 nodes and 63855 arcs. The number of nodes and arcs contained in the Strongly Connected Components (SCC) graph (Jensen 1997) is the same as in the state space. This implies that the state space has no cycles. So the INVITE transaction with reordering medium has no livelocks. The report also shows no dead transitions, so the INVITE transaction with reordering medium has no dead codes.

Analysis of the property of *absence of deadlocks*, however, is not straightforward. There are 8659 dead markings, which are too many to be all displayed by the CPN Tools (see Table 2, only five of the 8659 markings are displayed). Even the tool could display all the 8659 markings, it would be very time consuming to check each of them to see if it is a desirable terminal state. So we have to design our own CPN queries (Jensen et al. 2007) to find out whether or not the transaction has deadlocks without looking at the details of every dead marking.

From Figure 1, when the transaction completes both client and server transactions must be in the *Terminated* state. So we firstly create two CPN queries, Q1 and Q2 (Table 3) to divide the 8659 dead markings into two groups: one group containing

Table 2: Part of the state space report for the restricted CPN model with a reordering medium

Statistics	Occurrence Graph
	Nodes: 30982 Arcs: 63855 Secs: 148
Dead Markings	SCC Graph
	Nodes: 30982 Arcs: 63855 Secs: 3
Dead Transition Instances	8659 [9999,9998,9997,9996,9995,...]
Dead Transition Instances	None

Table 3: CPN Queries used in state space analysis

Name	Query	Output: Reordering medium	Output: Reordering & Lossy medium
Q1	size(SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)=0 or else cf(terminatedS, Mark.INVITE'Server 1 n) = 0), NoLimit, fn n => n, [], op :))	49 (Group 1)	372 (GroupL 1)
Q2	size(SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)>0 and also cf(terminatedS, Mark.INVITE'Server 1 n) > 0),	8610 (Group 2)	1220 (GroupL 2)
Q3	size(SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)=0 and also cf(terminatedS, Mark.INVITE'Server 1 n) = 0),	0	0
Q4	size(SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)=0 and also cf(terminatedS, Mark.INVITE'Server 1 n) > 0),	48 (Group 1a)	364 (GroupL 1a)
Q5	size(SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)>0 and also cf(terminatedS, Mark.INVITE'Server 1 n) = 0),	1 (Group 1b)	8 (GroupL 1b)
Q6	size(SearchNodes (ListDeadMarkings (), fn n => (cf(proceeding,Mark.INVITE'Client 1 n)>0 and also cf(terminatedS, Mark.INVITE'Server 1 n) > 0),	48	364
Q7	SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)>0 and also cf(terminatedS, Mark.INVITE'Server 1 n) = 0),	[3]	[5, 23, 145, 808, 2956, 7929, 17188, 23735]
Q8	size(SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)>0 and also cf(terminatedS, Mark.INVITE'Server 1 n) > 0 and also Mark.INVITE'Requests 1 n = empty and also Mark.INVITE'Responses 1 n=empty),	232 (Group 2a)	1220
Q9	size(SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)>0 and also cf(terminatedS, Mark.INVITE'Server 1 n) > 0 and also Mark.INVITE'Requests 1 n = empty and also Mark.INVITE'Responses 1 n <> empty),	1460 (Group 2b)	0
Q10	size(SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)>0 and also cf(terminatedS, Mark.INVITE'Server 1 n) > 0 and also Mark.INVITE'Requests 1 n <> empty and also Mark.INVITE'Responses 1 n=empty),	1140 (Group 2c)	0
Q11	size(SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)>0 and also cf(terminatedS, Mark.INVITE'Server 1 n) > 0 and also Mark.INVITE'Requests 1 n <> empty and also Mark.INVITE'Responses 1 n<>empty),	5778 (Group 2d)	0
Q12	size(SearchNodes (ListDeadMarkings (), fn n => (cf(terminated,Mark.INVITE'Client 1 n)>0 and also cf(terminatedS, Mark.INVITE'Server 1 n) > 0 and also Mark.INVITE'Requests 1 n <> empty and also (cf(INVITE,Mark.INVITE'Requests 1 n)>0),	2797	0

dead markings in which the client or server is terminated; and one containing dead markings in which both the server and client are terminated. With Q1, all dead markings (`ListDeadMarkings()`) are searched to find out the number of the dead markings whose Client place has zero terminated token (`cf(terminated,Mark.INVITE'Client 1 n)=0`) or whose Server place has zero terminatedS token (`cf(terminatedS, Mark.INVITE'Server 1 n)=0`). Note that the last four lines of Q1 show the other parameters required by the CPN function `SearchNodes`, which are the same for all queries in Table 3. To save space, the four lines are omitted from other queries. The result of Q1 (column 3 of Table 3) shows that among the 8659 dead markings, there are 49 markings in which the client or server is not terminated. Q2 searches the set of all dead markings for the number of markings in which both the client and server are terminated. The result of

this query is 8610. The sum of 49 and 8610 is 8659, number of all dead markings. So the set of all dead markings can be divided into two disjoint groups: **Group 1** with 49 markings and **Group 2** with 8610 markings.

We then investigate each of the two groups. Q3 checks if there are markings of Group 1 in which both the client and server are not terminated. The value returned is zero. Then we use Q4 to get the number of markings in which the client is not terminated and the server is terminated, and the number is 48. Q5 returns the number of markings in which the client is terminated but the server is not terminated, which is 1. The two numbers, 48 and 1, show that Group 1 can be divided into two disjoint groups: **Group 1a** (containing the 48 markings), and **Group 1b** (containing only one marking).

Interestingly, the result of Q6 shows that there are 48 dead markings in which `Client=proceeding`

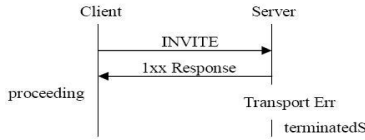


Figure 5: Illustration of the deadlock (Ding & Liu 2008)

and `Server=terminatedS`. Because Group 1a also comprises 48 dead markings, and for each of them `Client≠terminated` and `Server=terminatedS`, all Group 1a markings have `Client=proceeding` and `Server=terminatedS`. We then check the details of the 48 markings of Group 1a, and find that each of them is caused by a transport error at the server side, so the client cannot receive any responses and has to stay in the *Proceeding* state. The 48 markings are distinct from each other because when a transport error occurs at the server side, messages in the `Requests` and `Responses` places may be different, and the counter places `INVITE sent`, `r3xxResent`, and `r101sent` (a place added to the restricted model) have different numbers. We have discovered in (Ding & Liu 2008) that when the transport medium is reliable, the INVITE transaction has a deadlock in which the client transaction is proceeding and the server transaction is terminated, which is also caused by a transport error at the server side (Figure 5). This behaviour is not desirable as in this case the server transaction has been destroyed, so no responses can be received by the client transaction, thus it cannot move out of the *Proceeding* state. We have proposed to fix the problem by adding a timer to the *Proceeding* state so the transaction can transition to its *Terminated* state when the timer expires (Ding & Liu 2008).

Then Q7 is used to identify the marking in Group 1b. Marking 3 is returned. We find that in this marking the `Client` is `terminated`, and the `Server` is `Idle`. This is the acceptable dead marking discussed in (Ding & Liu 2008). It is caused by the transport error occurring at the client side when SIP transport layer sending the original INVITE request. So we do not consider this dead marking as a deadlock.

With Group 2, we use Q8, Q9, Q10 and Q11 to find out the numbers of markings in which places `Requests` and `Responses` may or may not empty. The value returned from each query is shown in column 3 of Table 3, and adding them up gives us 8610, total number of markings in Group 2. This indicates that Group 2 can be divided into four disjoint groups: **Group 2a**, **Group 2b**, **Group 2c** and **Group 2d**.

For each of the 232 marking of Group 2a, both places `Requests` and `Responses` are empty and both client and server are terminated, which are the desirable dead markings of the INVITE transaction. We have more than one desirable dead markings because the counter places `INVITE sent`, `r3xxResent`, and `r101sent` have different numbers, which is expected.

For each marking of Group 2b, only place `Responses` has messages. As discussed in (Ding & Liu 2008), RFC 3261 specifies that any responses arriving at the client side after the matching transaction has been destroyed will be discarded by SIP transport layer silently. So markings in Group 2b are acceptable as our model does not capture the behaviour of SIP transport layer of discarding the responses after the client is terminated.

However, for markings in Group 2c or Group 2d, there are messages in place `Requests`. Q12 is then used to check whether in these mark-

ings INVITE requests are in place `Requests` (`cf(INVITE,Mark.INVITE'Requests 1 n)>0`). The result is 2797 (instead of zero). This shows that for the 2797 markings of Group 2c and 2d, INVITE requests are left in `Requests` after the server is terminated. This is not desirable because according to RFC 3261, when the server side receives an INVITE request that does not match any active server transaction, the INVITE will be treated as a new INVITE request and a new server transaction is created. This problem is also identified by SIP implementers and it has been presented in a recently submitted Internet Draft by Sparks (2008). So our findings have provided a theoretical evidence to the problem presented in the Internet Draft.

In summary, the INVITE transaction with reordering medium is absence of livelocks and dead codes. It has deadlocks, including *Type 1 deadlocks*, the 48 dead markings of Group 1a; and *Type 2 deadlocks*, the 2797 dead markings of Group 2c and Group 2d in which place `Requests` has INVITE tokens.

4.4 Reordering and Lossy Medium

We use the same grouping approach to analyse the INVITE transaction with reordering and lossy medium (i.e. the restricted model). Because the occurrences of transitions `Lose Request` and `Lose Response` remove messages randomly from places `Requests` and `Responses`, there are more possible combinations of messages in the two places than in the case of a reordering only medium. Thus the state space has more markings (Table 4). From the state space report (Table 4), the INVITE transaction over a reordering and lossy medium is also absence of livelocks because the sizes of the state space and the SCC graph are the same (i.e. no cycles in the state space). It is absence of dead codes too as no dead transitions are reported.

In this case, we have 1592 dead markings, much less than the number of dead markings in the case of a reordering medium. However, it is still too many to check them one by one. So we again use the idea of grouping to gain insight into the dead markings. Queries Q1 to Q11 are used, and the values returned by the queries are shown in the last column of Table 3. We see that the 1592 dead markings can be firstly divided into two disjoint groups: **GroupL 1** and **GroupL 2**. GroupL 1 has 372 markings and in each of them the client or the server is not terminated (result of Q1); GroupL 2 has 1220 states and in each of them both the client and sever are terminated (result of Q2).

GroupL 1 can be divided into two groups: **GroupL 1a** with 364 markings, and in each of them the `Client` is `proceeding` (result of Q6); **GroupL 1b** with 8 markings, and in each of them the `Client` is `terminated` and the `Server` is `Idle` (result of Q5). As discussed in Section 4.3, markings of GroupL 1a are deadlocks.

We find that in each of the 1220 markings of

Table 4: Part of the state space report for the restricted CPN model

Statistics	Occurrence Graph	
	Nodes:	278031
Arcs:	1280815	
Secs:	24010	
Dead Markings	SCC Graph	
	Nodes:	278031
Arcs:	1280815	
Secs:	71	
Dead Transition Instances	None	

GroupL 2, places **Requests** and **Responses** are both empty, i.e. no messages left in the channels (see the result of Q8). (In fact when using another query to check the 372 markings of GroupL 1, we find that in each of them both places are also empty.) This is expected as occurrences of transitions **Lose Request** and **Lose Response** eventually destroy all the messages in the two places after the client and/or server are terminated, including the messages that have arrived at the server/client side after the server/client is terminated (thus should not be “lost” by the two transitions). This is a result of the modelling choice that we do not include our model the operations of TUs discarding unmatched responses (at the client side) or creating new server transactions as a result of receiving unmatched INVITE requests (at the server side). Therefore in this case we are not able to observe Type 2 deadlocks (Section 4.3), i.e. undesirable markings in which INVITE requests are left in the channel when the server is terminated. This also shows why we have taken an incremental approach to the analysis. If we did not firstly analyse the case when the medium reorders messages only, we would not detect Type 2 deadlocks.

5 Conclusions and Future Work

As a continuation of the work presented in (Ding & Liu 2008), in this paper, we have modelled and analysed SIP INVITE transaction over an unreliable medium that may reorder or lose messages. We firstly create a CPN model for the INVITE transaction over an unreliable medium. Then by examining the state space of the model, we have found that the INVITE transaction is free of livelocks and dead codes, as in the case of a reliable medium. Therefore we can conclude that the INVITE transaction is absence of livelocks and dead codes no matter over what medium it is operating. The state space analysis also reveals that when the medium is unreliable, the transaction has deadlocks in which the INVITE client transaction is in its *Proceeding* state (waiting for responses from the server side) while the server transaction is already terminated. In (Ding & Liu 2008), we also found similar deadlocks with the case when the medium is reliable. So we say that the INVITE transaction is not free of deadlocks with either a reliable or unreliable medium. Moreover, we have found that when the medium is unreliable, retransmitted INVITE requests can arrive at the server side when the matching server transaction has been terminated, which causes the TU at the server side to create new server transactions (but the client transaction that sent the INVITE requests has been terminated). This a problem with the INVITE transaction only when the medium is unreliable.

To carry out state space analysis for the INVITE transaction over an unreliable medium, we have firstly discussed and obtained a restricted model whose state space is tractable while the restrictions do not prevent us from investigating most scenarios in practice. When analysing the state spaces to detect deadlocks, we have to handle large number of dead markings. We have proposed the idea of *grouping* dead markings, to reduce search spaces so that we can gain insight into the dead markings as groups instead of enumerating every dead marking.

In this paper, based on the analysis of the relations between timers, we have captured the occurrence sequences of events of the INVITE transaction, without modelling the values of timers. Given that the INVITE transaction uses many timers, it would be interesting to model the values of timers using timed Coloured Petri Nets, so we can check properties related to timers, e.g. whether the values designated to

all timers are appropriate or not. Therefore in the future we would like to model and analyse the INVITE transaction using timed Coloured Petri Nets.

The second type of deadlocks (which only occur when medium is unreliable) is also presented by the recently submitted Internet draft (work in progress) (Sparks 2008). Our finding in this paper has provided a theoretical support to the Internet Draft about these deadlocks. As a further investigation, we would like to verify the updates to the INVITE transaction state machines proposed by the Internet Draft, to provide timely input to the development of the draft.

References

- Billington, J., Gallasch, G.E. & Han, B. (2004), Lectures on Concurrency and Petri Nets: A Coloured Petri Net Approach to Protocol Verification, *LNCS*, vol. 3098, pp. 210-290, Springer.
- Homepage of the CPN Tools, <http://wiki.daimi.au.dk/cpntools/cpntools.wiki>.
- Ding, L.G. & Liu, L. (2008), Modelling and Analysis of the INVITE Transaction of the Session Initiation Protocol Using Coloured Petri Nets, *in* ‘the 29th Int. Conf. on Applications and Theory of Petri Nets and Other Models of Concurrency’, *LNCS*, vol 5062, pp. 132-151, Springer.
- Gehlot, V. & Hayrapetyan, A. (2006), A CPN Model of a SIP-Based Dynamic Discovery Protocol for Webservices in a Mobile Environment, *in* ‘the 7th Workshop and Tutorial on Practical Use of CPNs and the CPN Tools’, Uni. of Aarhus, Denmark.
- Holzmann, G.J. (1991), *Design and validation of computer protocols*, Prentice Hall, New Jersey.
- Jensen, K. (1997), *Coloured Petri nets: Basic Concepts, Analysis Methods and Practical Use*, Vol. 1, 2, 3, 2nd edition, Springer.
- Jensen, K., Kristensen, L. & Wells, L. (2007), Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems, *Int. J. on Software Tools for Technology Transfer*, vol. 9, no. 3, pp. 213-254, Springer.
- Kristensen, L.M., Jørgensen, J.B. & Jensen, K. (2004), Application of Coloured Petri Nets in System Development, *in* ‘the 4th Advanced Course on Petri Nets’, *LNCS*, vol. 3098, pp. 626-685, Springer.
- Peng, Y., Yuan, Z. & Wang, J. (2007), Petri Net Model of Session Initiation Protocol and Its Verification, *in* ‘the Int. Conf. on Wireless Communications, Networking and Mobile Computing’, pp. 1861-1864, IEEE.
- Sun, W., Liu, F., Dai, G. & Li, H. (1997), *in* ‘the 8th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies’, pp. 488-492, IEEE.
- Rosenberg, J., et al. (2002), RFC 3261: SIP: Session Initiation Protocol. IETF, <http://www.faqs.org/rfcs/rfc3261.html>.
- Sparks, R. (2008), ‘draft-sparks-sip-invfif-02: Correct transaction handling for 200 responses to Session Initiation Protocol INVITE requests’, IETF, <http://tools.ietf.org/id/draft-sparks-sip-invfif-02.txt>.
- Wan, H., Su, G. & Ma, H., SIP for Mobile Networks and Security Model, *Wireless Communications, Networking and Mobile Computing*, pp.1809-1812, IEEE.