HAL
open science

Formal modeling and analysis of a narrow bandwidth protocol for establishing and terminating connections

Hai Wan, Ming Gu, Xiaoyu Song

To cite this version:

Hai Wan, Ming Gu, Xiaoyu Song. Formal modeling and analysis of a narrow bandwidth protocol for establishing and terminating connections. Mathematical and Computer Modelling, 2009, 50 (7-8), 10.1016/j.mcm.2009.06.001 . inria-00516025

HAL Id: inria-00516025

https://inria.hal.science/inria-00516025v1

Submitted on 13 Mar 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal Specification and Verification of a Narrow Bandwidth Protocol in PVS

Hai Wan[1,2], Ming Gu[2] and Xiaoyu Song[3]

[1] CST Dept, Tsinghua University, China
[2] School of Software, Tsinghua University, China
[3] ECE Dept, Portland State University, Portland, USA

Abstract. The paper presents a protocol for connection-establish service over an extreme-narrow bandwidth channel. The protocol is modeled and verified in a theorem proving system PVS. The PVS formalization and proofs of properties are performed. The execution behaviors of the protocol are modeled by state traces. Inductive methods were used to verify three important properties of the protocol. The three properties are held by the protocol for an arbitrary number of agents. The effectiveness of the approach is demonstrated by detection of a bug in the initial protocol implementation.

Key words: Connection establishment protocol, Narrow bandwidth, theorem proving, PVS

1 Introduction

Protocol verification technologies have progressed in recent years. A variety of tools are available for analyzing protocols. Due to the complexity of protocols, it is of paramount importance that protocol implementations be verified towards reliable systems. Communication protocols are usually modeled by a set of finite-state machines that generate the interaction between processes. Protocol verification is a procedure to validate the logical correctness of these interaction sequences and to detect potential design errors.

Model checking [7] is employed for verifying a protocol. A model checker works on a finite-state model of the system to be verified, and a logical specification of a desired behavior of the system model. It checks that the model adheres to the specification by effectively searching the entire state space of the model. On the other hand, theorem proving is envisioned as the most general approach to verification. With theorem proving approaches, the correctness condition of system is formalized as a theorem in a mathematical logic and a mechanically checked proof of theorems is generated using a general-purpose theorem proving system. The approach has attained significant successes in verifying digital system designs [2]-[6]. Theorem systems use powerful formalisms such as higher-order logic that allow the verification problem to be stated at many levels of abstraction.

In our system, connection establishment is of great importance, since major applications are available only after a connection is set up. Due to the limitations of the transfer channel, the connection-establish protocol should be enough, i.e., the steps token and bandwidth cost should be small. Classical connection-establish protocols such as three-way handshake [9] can not fit our requirements very well. In this paper, we present and verify a connection-establish protocol in PVS. We proved the protocol holds the properties.

Three-way handshake protocol had been studied in [1]. Our approach differs from theirs in several aspects: protocols being verified are different, transfer channels that the protocol based on, the modeling of timeout and our approach consider arbitrary pairs of participants.

PVS [8] is a mechanized environment for formal specification and verification. PVS consists of an input language, a type checker, and an interactive prover. The specification language of PVS is based on the classical typed higher-order logic. With the expressive language, the protocol can be translated into the PVS codes, and the properties need to prove can be written as theorems. The powerful primitive inference procedures provided by PVS can be used to reduce the proof effort significantly.

The rest of the paper is organized as follows. In Section 2, a detailed description about the system is given. Section 3 presents the protocol and the properties must be hold. We show how we modeled the protocol and proved the properties using PVS in Sections 4 and 5, respectively. Section 6 concludes the paper.

2 The Narrow Bandwidth Protocol

We describe the system and the protocols. The architecture of the system we consider is shown in Fig 1. The system is composed of a transfer channel and a network of agents. The transfer channel provides a fundamental and very low level message deliver service (MDS for short in the sequel). Every agent connecting to the channel can use MDS to communicate with any other agent by exchanging messages. Because of the underlying hardware on which MDS is implemented, MDS has the following limitations and characteristics:

1. The bandwidth of MDS is very limited, i.e. the typical bandwidth of the channel is about fifty bytes per ten seconds. In other words, one agent can only send fifty bytes at one time, and the time span between every two adjacent send-actions of one agent must be longer than 10 seconds.
2. MDS is not reliable. It may lose messages when transferring.
3. Message delivery may delay, i.e., assuming that agent1 sends a message M to agent2 and agent2 receives M, then agent2 must receive M within a time bound.
4. One agent can send at most one message at one time.
5. The order of received messages is correspond with the order of messages sent, i.e. if an agent receives message M1 before M2, then message M1 must be sent prior to M2. MDS is similar to the cell phone's short messages service.

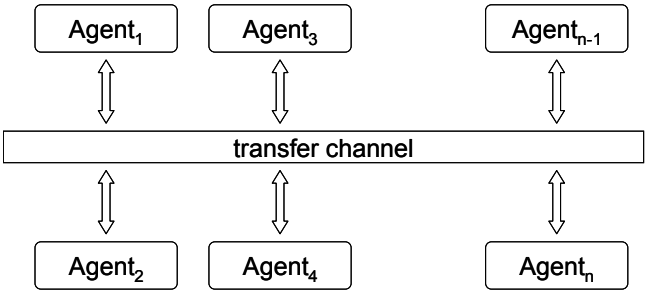

Fig. 1. The architecture of the system

In our system, agents need to exchange long length content such as files and long messages (longer than fifty bytes). MDS cannot provide such service directly because of its limitations. In order to solve this problem, we established a protocol stack which consists of two layers: extreme-narrow-bandwidth-connection protocol (ENBCP) and long-message-send protocol (LMSP). ENBCP is used to establish and terminate a connection between two agents. After ENBCP establishes a connection between two agents, which means they can communicate with each other through an exclusive channel, any one of the two can transfer long length content using LMSP. When finishing transferring, agents can use ENBCP to terminate the connection. The relation between ENBCP and LMSP is demonstrated in Fig 2. There are a number of P to P transfer protocols that

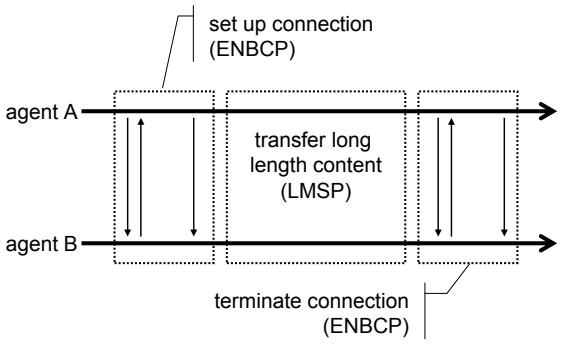

Fig. 2. ENBCP and LMSP

can be used as LMSP with a few modifications. We do not discuss the modeling and verification of these protocols in this paper. MDS requires that ENBCP be as simple as possible. In other words, ENBCP should take as less steps as possible, while the message length of each step should be as short as possible.

Protocols, such as three-way handshake of TCP/IP, are not suitable for this situation. It takes long time to establish a connection and the length of one frame of TCP/IP might exceed the fifty bytes' limitation of MDS. ENBCP should be reliable, since it is the basis of LMSP. We developed an ENBCP, which takes four steps to establish and terminate a connection. In each step, only 4 bits are used by ENBCP for identifying the message kind and the rest bits can be used for storing the initialization data of LMSP, etc.

In ENBCP, the agent which establishes a connection is called sender and the other part of the connection is called receiver. An agent can take only one role (sender or receiver) at a time. A agent has five states: *Idle*, *BeforeSend*, *Sending*, *BeforeSendEnd* and *Recving*. A sender agent has four states: *Idle*, *BeforeSend*, *Sending* and *BeforeSendEnd*. A receiver agent has two states: *Idle* and *Recving*. There are four types of messages in ENBCP: *SendReq*, *SendRes*, *SendEndReq* and *SendEndRes*.

The pseudo-codes of ENBCP are shown in Fig 3. *MQ* is a message queue, whose items are of the form (*from*, *type*). If MDS receives a message, it will automatically add an item to *MQ*. The field *from* is the id of the agent that sent the message, while the field *type* is the message's type. Given a message *M*, we can access the field from by *M.from*. *DQ* is a demand queue whose items are added by the agent. Items of *DQ* are of the form (*with*, *type*). The field *with* is the id of the agent with whom it wants to establish a connection. There are two types of demands: *EstablishConnection* and *TerminateConnection*. For example, if an agent wants to establish a connection with agent *B*, it adds an item (*B*, *EstablishConnection*) to its *DQ*. Variable state denotes for the agent's state and variable *cw* denotes for the agent with whom it wants to establish a connection or already has established a connection. Function *Send(a, t)* denotes for sending a message of type *t* to agent *a*.

ENBCP consists of three parts:

1. State transition in terms of the messages received.
   If an agent receives a *SendReq* and its state is *Idle*, then it changes its state to *Recving* and send a response message *SendRes*.
2. State transition according to the demands.
   After picking a *EstablishConnection* demand from DQ and verifying the condition (state = *Idle* and no *SendReq* message in MQ), the agent sends a *SendReq* message and sets its *cw* accordingly.
3. Timeout testing.
   In ENBCP, an agent cannot stay at any *non-Idle* state for an arbitrary time. There are timeout thresholds for every *non-Idle* state. If an agent stays at *Recving* state for a time longer than *time_out_Recving*, then it changes its state back to *Idle*.

The state transition graph of the agent is shown in Fig 4. Messages sent by the agent are marked with a "!" symbol. Messages received by the agent are marked with a "?" symbol. The solid and dashed arrows denotes for the state transitions result in message send/receiving actions or timeout respectively.

```
MQ : Message Queue
DQ : Demand Queue
state : Agent State
cw : Communication width
MP = Empty; DP = Empty; state = Idle; cw = None
loop forever
    if MQ is not Empty then get a message M from MQ;
    case M.type of
        SendReq: if state = Idle then
                    cw = M.from; Send(cw, SendRes); state = Recving;
        SendEndReq: if state = Recving and cw = M.from then
                    Send(cw, SendEndRes); state = Idle; cw = None;
        SendRes: if state = BeforeSend and cw = M.from then
                    state = Sending; (notify the connection is established)
        SendEndRes: if state = BeforeSendEnd and cw = M.from then
                    state = Idle; cw = None; (notify connection terminates normally)
    end case

    if DQ is not Empty then get a demand D from DQ;
    case D.type of
        EstablishConnection: if state = Idle and (no SendReq message in MQ) then
                        cw = D.with; Send(cw, SendReq); state = BeforeSend
        TerminateConnection: if state = Sending then
                            Send(cw, SendEndReq); state = BeforeSendEnd
    end case

    case state of
        Recving : if stays in state Recving longer than time_out_Recving then
                        state = Idle; cw = None;
        BeforeSend: if stays in state BeforeSend longer than time_out_BeforeSend then
                        state = Idle; cw = None; (notify connection terminates abnormally)
        Sending: if stays in state Sending longer than time_out_Sending then
                        state = Idle; cw = None; (notify connection terminates abnormally)
        BeforeSendEnd: if stays in state BeforeSendEnd exceeds time_out_BeforeSendEnd then
                        state = Idle; cw = None; (notify connection terminates abnormally)
    end case
end loop
```

Fig. 3. The pseudo-code of ENBCP

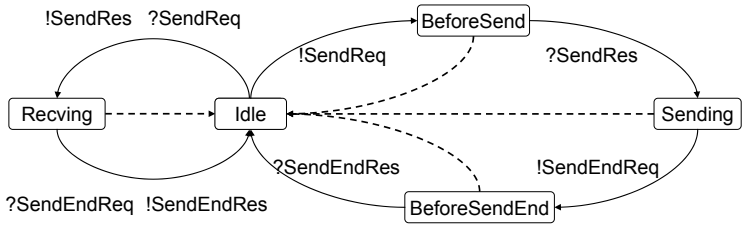

Fig. 4. The state transition graph

A typical execution of the protocol is shown in Fig 5. The states of the agent are marked above the lines and the messages of send and receive are labeled by the arrows. It needs only two messages for establishing and terminating a connection. The execution steps are: 1) if $agent_1$ wants to establish a connection with $agent_2$, first it makes sure its state is *Idle* and does not receive any *SendReq* in *MQ*, then it sends *SendReq* to $agent_2$; 2) if $agent_2$ receives *SendReq* and its state is *Idle*, then it sends *SendRes* to $agent_1$; 3) after $agent_1$ receives *SendRes* from $agent_2$, which means it has establish a connection with $agent_2$, it can start LMSP with $agent_2$; 4) after executing LMSP, $agent_1$ sends *SendEndReq* to $agent_2$; 5) if $agent_2$ receives *SendEndReq* from $agent_1$, then it sends *SendEndRes* to $agent_1$; 6) $agent_1$ receives *SendEndRes* from $agent_2$, which means the connection is terminated.

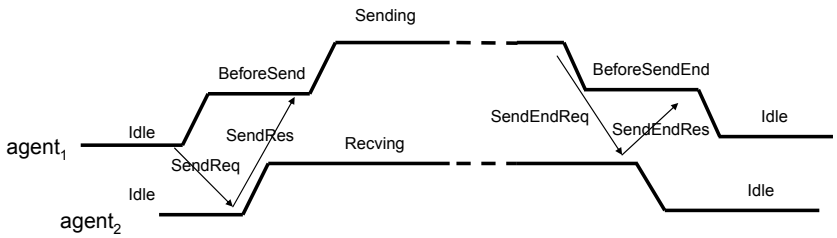

Fig. 5. A typical execution of ENBCP

There are several properties that ENBCP should hold to guarantee the correctness.

Property 1. If an agent A's state is *Sending*, then there must be another agent B, while B's state is *Recving*, B's id is equal to A's *cw* and B's *cw* is equal to A's Id.

An agent A is said to be at *Ready* mode with agent B if and only if the following conditions hold: 1) A's state is *Sending*; 2) A'*cw* is B's *Id*; 3) B's state is *Recving*; 4) B's *cw* is A's *Id*.

It's obvious that when an agent is at *Ready* mode with some other agent, it can start the LMSP protocol safely. The first two conditions are easily verified, since it only refers to local variables. From property 1 we can infer that the first two conditions imply the last two conditions. So the *Ready* mode check can be checked locally.

Property 2. At any time, two different agents are not allowed to be at the Ready mode with the same agent. In other words, if agent A is at *Ready* mode with agent C and agent B is at *Ready* mode with agent B, then A is equal to B.

Property 3. Any agent whose state is non-*Idle* will eventually reach *Idle* state. This property expresses the deadlock-free of ENBCP.

3 Modeling Protocol Primitives

We elaborate on three theories to model the protocol. Theory *rt* contains the fundamental primitives about time, theory *env* specifies the model of the transfer channel and theory *protocol* defines the model of the protocol.

3.1 Formalizing Time

We define a theory *rt* for time. We model time as type *int* in PVS. Based on Type *Interval*, variant time intervals are defined, such as *co* represents left-closed and right-open intervals, etc. Predicate *during* is defined as a moment in an interval and predicate *before* as a moment before the other.

```
Time        : TYPE = int
PTime       : TYPE = {t:Time|t>0}
Interval        : TYPE = pred[Time]
t,t0,t1         : VAR Time
cc(t0,t1)       : Interval = {t|t0<=t AND t<=t1}
co(t0,t1)       : Interval = {t|t0<=t AND t<t1}
oc(t0,t1)       : Interval = {t|t0<t AND t<=t1}
oo(t0,t1)       : Interval = {t|t0<t AND t<t1}
during(t, I)        : bool = I(t)
before(t0, t1)  : bool = t0 < t1
```

3.2 Formalizing Message Deliver Services

Enumerative types *StatusType* and *MsgType* are used to represent the states of an agent and the types of messages respectively. *Msg* which is a record type has one field: *MsgType*. The constant *NoUser* denotes for *None* in ENBCP. All other agents are represented by the type *User*. The agents in the protocol are of type *AnyUser* in the model.

```
MsgType : TYPE = { SendReq, SendRes, SendDataReq,
    SendDataRes, SendEndReq, SendEndRes}
```

```
StatusType : TYPE = { Idle, BeforeSend, Sending, Recving,
BeforeSendEnd}
Msg : TYPE = [#mtype : MsgType #]
AnyUser : TYPE+ % = [# stype : StatusType #]
NoUser : AnyUser
User : TYPE+ = {u:AnyUser|u /= NoUser}
```

The channel transfer delay and four time out thresholds are list below. We should mention that the assumptions on these constants such as $t_timeout > 2 * sr_delay$ are necessary for proving the first and second properties.

```
sr_delay : PTime            % transfer delay
send_interval : PTime           % interval between
                                %two send actions
t_timeout : {t:PTime|t>2*sr_delay}
            % time_out_BeforeSend and time_out_BeforeSendEnd
t_windowtimeout_sender : {t:PTime|t>2*t_timeout}
            % time_out_Sending
t_windowtimeout_receiver :
            {t:PTime |t>t_windowtimeout_sender+t_timeout}
            % time_out_Recving
```

The *send* and *receive* actions are modeled by two functions *Send* and *Recv* respectively. Predicate *Send(u1,m,t,u2)* specifies the fact that user *u1* sends a message *m* to user *u2* at time *t*. *Recv(u1,m,t,u2)* means that user *u1* receives a message *m* from user *u2* at time *t*.

```
Send, Recv : [User, Msg, Time, User -> bool]
```

We devise a list of axioms as constraints for the *send* and *receive* actions to model characteristics of MDS.

```
OneUserCanSendOnlyOneMsgAtOneTime : AXIOM
    FORALL (u,u1,u2,m1,m2,t) :
        Send(u,m1,t,u1) AND Send(u,m2,t,u2)
            IMPLIES (m1 = m2) AND (u1 = u2)
RecvOrder0 : AXIOM
    Send(u1,m1,t1,u2) AND Send(u3,m2,t2,u4) AND
        before(t1, t2)AND Recv(u2,m1,t3,u1) AND
        Recv(u4,m2,t4,u3) IMPLIES before(t3, t4)
SendInterval : AXIOM
    Send(u,m1,t1,u1) AND Send(u,m2,t2,u2) AND before(t1, t2)
    IMPLIES before(t1, t2 - send_interval)
RecvBySend : AXIOM
    Recv(u1,m,t2,u2) IMPLIES
       EXISTS (t1) : (Send(u2,m,t1,u1) AND
       during(t2, oc(t1, t1 + sr_delay)))
```

```
OneMsgCanBeSendOnce : AXIOM
    FORALL (u1,u2,u3,u4,m,t1,t2) :
        Send(u1,m,t1,u2) AND Send(u3,m,t2,u4)
        IMPLIES (u1 = u3) AND (t1 = t2) AND (u2 = u4)
MsgCanBeRecvOnce : AXIOM
    Recv(u2,m,t1,u1) AND Recv(u4,m,t2,u3)
    IMPLIES (u1 = u4) AND (t1 = t2) AND (u1 = u3)
```

Axiom *OneUserCanSendOnlyOneMsgAtOneTime* implies that one user can send only one message at one time. Axiom *RecvOrder0* describes the message-ordering characteristic of MDS (the fifth characteristic of MDS in section 2). Axiom *SendInterval* describes the fact that the time span between two adjacent send-actions of one agent must be longer than the constant *send_interval*. Axiom *RecvBySend* means that if user *u1* receives a message *m* from user *u2* at time *t2*, there must be a time *t1*, on which *u2* sent the message *m*. And *t1* and *t2* satisfy the deliver delay constraint. The last two axioms are auxiliary.

4 Formalizing the Narrow Bandwidth Protocol in PVS

We model the connection protocol in the theory *protocol*. The status of an agent is modeled by a record type *OneStatus* of two fields *StatusType* and *User*. *cw* of type *User* denotes the user with which the agent is communicating.

```
OneStatus : TYPE = [# st : StatusType, cw : User #]
```

Type *OneUserTrace* denotes for the status at a certain point of time. Every item in *OneUserTrace* is a status trace with respect to time. Given an item *out* of type *OneUserTrace* and a time *t*, *out(t)* denotes for the status at time t. Any element in type *Traces* is a function maps from *User* to *OneUserTrace*. Given an item *T* of type *Traces*, a *User u* and a time *t*, *T(u)(t)* denotes for the *status* of agent *u* at time *t*.

```
OneUserTrace : TYPE = [Time -> OneStatus]
Traces : TYPE = [User -> OneUserTrace]
```

Since the elements of *Traces* are arbitrary and we don't put any constraints on them, it is possible that, given a *T* of *Traces*, a *u* of *User* and a *t* of *time* that T(u)(t).*StatusType* = *Sending* and *T(u)(t+1).StatusType* = *Recving*, which obviously disobeys the ENBCP. In order to avoid this violation, we devise sorts of constraints (predicates) about *Traces* based on ENBCP. There are five sorts of predicates.

4.1 Formalizing State Transitions

This type of predicate describes the valid changes between states. Take predicate *VSC_Idle* for example; it denotes that after state *Idle*, the valid states next time are *Idle*, *BeforeSend* or *Recving*. The predicates about *BeforeSend*, *Sending*, *Recving* and *BeforeSendEnd* are of the same form.

```
VSC_Idle(outr) : bool =
        FORALL (t) : (outr(t)`st = Idle) IMPLIES
            (outr(t+1)`st = Idle) OR (outr(t+1)`st = BeforeSend) OR
            (outr(t+1)`st = Recving)
```

4.2 Formalizing Timeout

When the time span one user stays at any non-*Idle* state exceeds the timeout threshold, the state will change to *Idle* next time. For instance, if one agent stays at state *BeforeSend* for a time span equal to *t_timeout*, its state will change to *Idle* or *Sending*.

```
VSC_BeforeSendTimeout(outr) : bool =
    FORALL (t1) :BeingStatus(outr,t1-t_timeout+1,t1,BeforeSend)
        IMPLIES
            outr(t1+1)`st = Sending OR outr(t1+1)`st = Idle
```

4.3 Formalizing Communicate Width

In the protocol, each user has variable *cw* to record the user with whom he communicates. The following predicates represent the constraints on the valid change of *cw* are given in the model.

```
VSWC_Idle(outr) : bool =
    FORALL (t) : (outr(t)`st = Idle) IMPLIES (outr(t)`cw = NoUser)
VSWC_ChangeWith(outr) : bool =
    FORALL (t) :
        (outr(t+1)`cw /= outr(t)`cw) IMPLIES (outr(t+1)`st /= outr(t)`st)
VCWC_IdleToBeforeSend(outr) : bool = FORALL (t) : (outr(t)`st =
Idle) AND (outr(t+1)`st = BeforeSend)
        IMPLIES
            outr(t+1)`cw /= NoUser
VCWC_BeforeSendToSending(outr) : bool =
    FORALL (t) : (outr(t)`st = BeforeSend) AND (outr(t+1)`st = Sending)
        IMPLIES
            outr(t)`cw = outr(t+1)`cw
VCWC_SendingToBeforeSendEnd(outr) : bool =
    FORALL (t) : (outr(t)`st = Sending) AND (outr(t+1)`st = BeforeSendEnd)
        IMPLIES
            outr(t)`cw = outr(t+1)`cw
VCWC_IdleToRecving(outr) : bool =
    FORALL (t) : (outr(t)`st = Idle) AND (outr(t+1)`st = Recving)
        IMPLIES
            outr(t+1)`cw /= NoUser
```

4.4 Formalizing Transfer Events

Every send action has its conditions and effects. For instance, when a send *SendReq* action takes place at time t, we can infer that the agent's state is *Idle* and it doesn't receive *SendReq* at time t and the agent's state is *BeforeSend* at *time (t+1)*. We use condition and effect to specify send actions.

```
VA_Send_SendReq(u,tr) : bool =
        FORALL (t,u1,m) :
            Send(u,m,t,u1) AND (m`mtype = SendReq)
            IMPLIES
(tr(u)(t)`st = Idle) AND (NOT EXISTS (u2) : (Recv(u,m,t,u2) AND
(m`mtype = SendReq))) AND (tr(u)(t+1)`st = BeforeSend) AND
(tr(u)(t+1)`cw = u1)
```

In the above specification, "$(tr(u)(t)`st = Idle)AND(NOTEXISTS(u2) : (Recv(u, m, t, u2)AND(m`mtype = SendReq)))$" is the condition of the send action, and "$(tr(u)(t + 1)`st = BeforeSend)AND(tr(u)(t + 1)`cw = u1)$" is the effect of the action. The specifications about *SendEndReq*, *SendRes* and *SendEndRes* are of the same form and are omitted here.

Based on the protocol, we infer that the state of an agent changes if and only if one of the following conditions holds: 1) The agent stays at non-Idle state exceeds the timeout threshold 2) The agent sends a message 3) The agent receives a message.

Predicate *VSCC_Idle_BeforeSend* denotes that if agent u's state changes from *Idle* to *BeforeSend* at time t, then u must send a *SendReq* message at time t.

```
VSCC_Idle_BeforeSend(u,tr) : bool =
        FORALL (t) :
            (tr(u)(t)`st = Idle) AND (tr(u)(t+1)`st = BeforeSend)
            IMPLIES
            (EXISTS (u1,m) : Send(u,m,t,u1) AND m`mtype = SendReq)
```

Predicate *VSCC_Recving_Idle* denotes that if agent u's state changes from *Recving* to *Idle* at time t, then u must receive a *SendEndReq* message at time t from agent cw or u has been at *Recving* state for a time span longer than the timeout threshold.

```
VSCC_Recving_Idle(u,tr) : bool =
    FORALL (t) :
            (tr(u)(t)`st = Recving) AND (tr(u)(t+1)`st = Idle)
        IMPLIES
            ((EXISTS (u1,m) : Recv(u,m,t,u1) AND m`mtype = SendEndReq
            AND tr(u)(t)`cw = u1)
                OR
            BeingStatus(tr(u),t-t_windowtimeout_receiver+1,t,Recving))
```

We have defined this kind of predicates for all the valid state change (*BeforeSend* to *Sending*, *Sending* to *Idle*, *BeforeSend* to *Idle*, ...).

5 Mechanically Proved Properties

We formalize the three properties described in Section 2 as follows.

Property 1. If an agent A's state is *Sending*, then there must be another agent B such that B's state is *Recving*, B's *id* equals A's *cw* and B's *cw* equals A's *Id*.

```
FinalTheorem_1 : THEOREM
    ValidTrace(tr) IMPLIES FORALL (t,u1) :
    tr(u1)(t)`st = Sending IMPLIES EXISTS (u2) : tr(u1)(t)`cw = u2 AND
    tr(u2)(t)`st = Recving AND tr(u2)(t)`cw = u1
```

ValidTrace(tr) denotes that *tr* is a trace satisfying all the five sorts of predicates described in section 3. Theorem *FinalTheorem_1* shows that if an agent *u1* is at *Sending* state at time *t*, then at time *t* there must exist an agent *u2*, whose state is *Recving* and with which *u1* establish connection to.

Property 2. At any time, two different agents are not allowed to be at the *Ready* mode with the same agent. In other words, if agent A is at *Ready* mode with agent C and agent B is at *Ready* mode with agent B, then A is equal to B.

```
FinalTheorem_2 : THEOREM ValidTrace(tr)
    IMPLIES FORALL (t,u1,u2,u3):
        tr(u1)(t)`st = Sending AND tr(u2)(t)`st = Sending AND
        tr(u1)(t)`cw = u3 AND tr(u2)(t)`cw = u3
        IMPLIES u1 = u2
```

Theorem *FinalTheorem_2* shows that, if two agents both are at Sending state and communicate with the same agent, then the two users must be equal.

Property 3. Any agent with state non-*Idle* will eventually reach *Idle* state. This property expresses the deadlock-freeness of ENBCP.

```
FinalTheorem_3 : THEOREM ValidTrace(tr)
    IMPLIES FORALL (t,u) : tr(u)(t)`st /= Idle
        IMPLIES
        EXISTS (t1) : before(t, t1) AND tr(u)(t1)`st = Idle
```

The theorem *FinalTheorem_3* is straightforward. To prove the first two theorems, we need prove the following theorem.

```
BeforeSendingExist : THEOREM ValidTrace(tr)
    IMPLIES FORALL (t) :
    tr(u)(t)`st = Sending IMPLIES EXISTS (t0,m1,m2,u1,t1,t2) : t2 < t
    AND BeingStatus(tr(u),t2+1,t,Sending) AND tr(u)(t2)`st = BeforeSend
    AND  t0 <= t2 AND BeingStatus(tr(u),t0,t2,BeforeSend) AND
    tr(u)(t0-1)`st = Idle AND t0<=t1 AND t1<=t2 AND Send(u1,m2,t1,u) AND
    m2`mtype = SendRes AND Recv(u,m2,t2,u1) AND u1 = tr(u)(t)`cw AND
    Send(u,m1,t0-1,u1) AND m1`mtype = SendReq AND Recv(u1,m1,t1,u) AND
    (t - t0 <= t_timeout + t_windowtimeout_sender) AND
    BeingStatus(tr(u1),t1+1,t,Recving)
```

The theorem can be illustrated in Fig 6. Given a user *u1* and a time *t*, if *u1*'s state is *Sending* at *t*, then we can infer the following conclusions:

1. There exists three times *t0*, *t1*, *t2* and a user *u2*;
2. *u1*'s state is *Idle* at time *t0*, *u1*'s state is *BeforeSend* between $t0+1$ and *t2*, *u1*'s state is *Sending* between $t2+1$ and *t*;
3. *u2*'s *cw* is *u1* and *u1*'s *cw* is *u2* at time *t*;
4. *u1* sends a *SendReq* to *u2* at *t0*, *u2* receives the request at *t1* and sends a *SendRes* to *u1* at *t1*, and *u1* receives the response at *t2*;
5. *u2*'s state is *Recving* between $t1+1$ and *t*;
6. Time span from *t0* to *t* is no more than $t_timeout + t_windowtimeout_sender$.

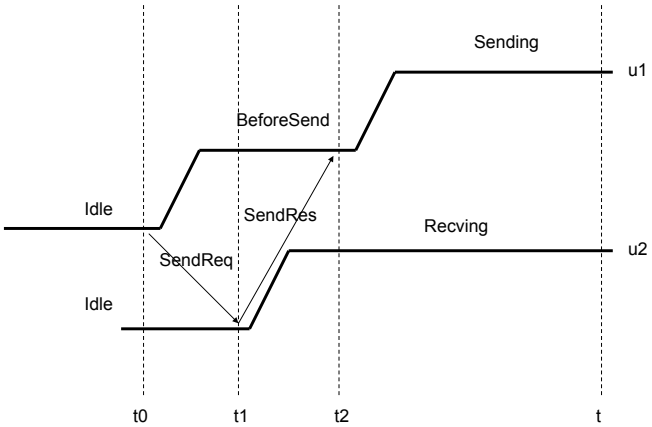

Fig. 6. The theorem BeforeSendingExist

To show theorem *BeforeSendingExist*, we need to prove 15 lemmas. The proof process is tedious and lengthy, we omit them here. Theorem *FinalTheorem_1* can be implied directly from the theorem *BeforeSendingExist*. To prove theorem *FinalTheorem_3*, we need to prove the following four lemmas based on the timeout mechanism of ENBCP.

- **Lemma 1.** If an agent's state is *Recving*, it will eventually reach *Idle*.
- **Lemma 2.** If an agent's state is *BeforeSendEnd*, it will eventually reach *Idle*.
- **Lemma 3.** If an agent's state is *Sending*, it will eventually reach *Idle* or *BeforeSendEnd*.
- **Lemma 4.** If an agent's state is *BeforeSend*, it will eventually reach *Idle* or *Sending*.

For property 2, from theorem *BeforeSendingExist*, we know that there must be two moments: *t1* and *t2*, as shown in Fig 7. There are three relations between

t1 and *t2*: $t1 > t2$, $t1 < t2$ and $t1 = t2$. Under the condition of $t1 = t2$, we can prove that $u1 = u2$, since one agent can send only one message at a time (*u3* at time *t1*). In case $t1 < t2$, from *BeforeSendingExist* we can infer that *u3*'s state must be *Recving* (for *u1*) and *Idle* (for *u2*), which causes a contradiction. The case $t1 > t2$ is similar.

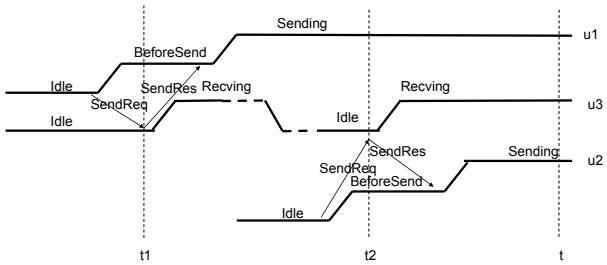

Fig. 7. Proof of FinalTheorem_2

Our proof detected a bug of the initial ENBCP protocol design. In the initial version of ENBCP, the constant *t_windowtimeout_receiver* is assumed to be larger than the constant *t_windowtimeout_sender*. During the proof process we found this assumption is too weak to prove the theorem BeforeSendingExist. After carefully study on the sub goals that cannot be proved, we strengthened this assumption to $t_windowtimeout_receiver > t_windowtimeou_sender + t_timeout$, then everything was done.

The assumptions about the delay and timeout thresholds are necessary for the correctness of the properties. But these assumptions are qualitative; the efficiency of the protocol entails further investigation. Timed-automata can be used to model and analyze the protocol and the transfer channel.

6 Conclusions

In this paper we presented a protocol of establishing connections on an extremely narrow bandwidth transfer channel. We modeled the protocol in the theorem proving system PVS. The execution behaviors of the protocol are modeled by state traces. The proof process detected a bug of the initial protocol implementation. Inductive methods were used to verify three important properties of the protocol. The three properties are held by the protocol for an arbitrary number of agents.

References

1. Schwabe, D.: Formal specification and verification of a connection establishment protocol. Proceedings of the seventh symposium on Data communications, pp.11–26, October 27-29, Mexico City, Mexico (1981)

2. Hooman, J.: Compositional Verification of a Distributed Real-Time Arbitration Protocol. (1994)
3. Rusu, V.: Verifying a Sliding Window Protocol using PVS. 21st International Conference on Formal Techniques for Networked and Distributed Systems, pp.251–268 (2001)
4. Dang, V.H.: Modelling and Verification of Biphase Mark Protocols in Duration Calculus Using PVS. Proceedings of the International Conference on Applications of Concurrency to System Design (1998)
5. Chkliaev, D., Stok, P.v.d., Hooman, J.: Formal Modeling and Analysis of Atomic Commitment Protocols. Proceedings of the Seventh International Conference on Parallel and Distributed Systems (2000)
6. Akbarpour, B.,Tahar, S.: Error Analysis of Digital Filters using Theorem Proving. In: K. Slind, A. Bunker and G. Gopalakrishnan (Eds.), Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science 3223, Springer Verlag, pp. 1–16. (2004)
7. Clarke, E., Grumberg, O., Peled, D.: Model Checking, MIT Press (2000)
8. PVS specification and verification system, http://pvs.csl.sri.com/index.shtml
9. Sunshine,C.A., Dalal,Y.K.: Connection Management in Transport Protocols. Computer Network 2, (6), December, (1978)