Stabilizing client/server protocols without the tears

Mohamed G. Gouda
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188
U. S. A.
Phone: 512-471-9532
Fax: 512-471-8885
Email: gouda@cs.utexas.edu

Abstract

We identify a rich class of client/server protocols. Each protocol in this class consists of a client and a server that engage in an infinite sequence of sessions. Each session consists of a bounded number of phases. In each even phase, the client sends a bounded number of messages to the server, and in each odd phase, the server sends a bounded number of messages to the client. We discuss a simple method for making each protocol in this class stabilizing. The method consists of three steps: session identification, session abortion, and session restart. The resulting protocols, thanks to their stabilizing properties, possess a high degree of fault tolerance.

Keywords

Client/server, convergence, fault-tolerance, network protocols, stabilization.

1 INTRODUCTION

A network protocol is stabilizing if it satisfies the following two conditions. First, if the protocol is in a safe state, then its execution keeps it in safe states. Second, if the protocol ever reaches an unsafe state, due to some fault occurrence, then its execution eventually leads it to a safe state. Clearly, stabilization provides network protocols, and computing systems in general, with a high degree of fault-tolerance.

The theory of system stabilization is outlined in (Gouda, 1995), the relationship between stabilization and fault tolerance is examined in (Arora and Gouda, 1994), and a comprehensive study of the stabilization of network protocols is reported in (Gouda and Multari, 1991).

In this paper, we identify a class of client/ server protocols and show how to make each protocol in this class stabilizing. Our objective of this exercise is three-fold. First, we hope to demonstrate to the reader that protocol stabilization is not only an intellectual curiosity but also a practical method for achieving fault tolerance. Second, we want to acquaint the reader with the method that we used to make our client/server protocols stabilizing. We feel that this method is important and can be used to make other protocols stabilizing. Third, we wish to convince the reader that his next client/server protocol should be stabilizing as defined in this paper.

Because protocol definitions occupy most of this paper, we devote the rest of this section to explain our notation for defining client/server protocols. A client/server protocol consists of two processes: a client process and a server process. Each process, whether a client or a server, is defined by a set of global constants, a set of local variables, and a set of actions as follows.

process <process name>
const <declarations of global constants>
var <declarations of local variables>
begin
<action> [] ... [] <action>
end

Each action in a process is of the form:

<guard> --> <statement>

where the guard is either a boolean expression or a receive statement of the form: **rcv** <message> **from** <process name>, and the statement is defined recursively as one of the following: skip, assignment, send, sequential composition, and conditional construct. These five forms of statements are discussed next.

A skip statement is of the form **skip**. This statement is executed by doing nothing.

An assignment statement is of the form:

<variable>, ... , <variable> := <expression>, ... , <expression>

This statement is executed in two steps. First, the value of each expression on the right-hand side is computed. Second, the computed values are assigned in parallel to their corresponding variables on the left-hand side.

2 CLIENT/SERVER PROTOCOLS

Consider a client/server protocol where the client and server engage in an infinite sequence of sessions. Each session consists of an even number of phases, and the number of phases in a session is at most m+1, for some odd integer m. In phase 0, the client sends at most n message to the server. In phase 1, the server sends at most n messages to the client. In phase 2, the client sends at most n messages to the server, and so on.

In the phase before last in a session, the client sends one message, an mrqst message, to the server requesting that the server commits all its work in this session. When the server receives the mrqst message, it executes the last phase in the session by committing its work in the session then sending one message, an mrply message, to the client. When the client receives the mrply message, it executes phase 0 in the next session by sending at most n messages to the server, and the cycle repeats.

Each sent message has a boolean field b. Field b in a message msg(b) is true iff msg(b) is the last message to be sent in its phase. By definition, the two messages mrqst and mrply are the last (and also first) messages to be sent in their phases. Hence, there is no need to include a boolean field b (whose value would always be true) in either message.

The client process has three variables phs, seq, and b. Variable phs stores the current phase in the current session, variable seq stores the number of message sent in the current phase, and variable b stores the field of the last received message. The client process can be defined as follows.

```
process client

const m, n      :   integer
                    {m+1 is the max. # of phases in a session}
                    {n is the max. # of messages sent in a phase}
                    {m is odd; m ≥ 3; n ≥ 1}

var   phs       :   0..m,
      seq       :   0..n,
      b         :   boolean

begin
      EVEN.phs ^ phs < m-1 -->
            if seq < n-1   -->   send msg(false) to server;
                                 seq := seq + 1
```

```
            [] true        -->   send msg(true) to server;
                                 phs := phs + 1
            fi

[]      rcv msg(b) from server -->
            if ~b --> skip
            []  b --> phs, seq := phs + 1, 0
            fi

[]      EVEN.phs -->
            send mrqst to server; phs := m

[]      rcv mrply from server -->
            {work of current session is committed}
            phs, seq := 0, 0
end
```

The client has four actions. In the first action, it sends a msg(b) message in an even phase, and in the second action, it receives a msg(b) message in an odd phase. In the third action, the client sends an mrqst message in the phase before last in the current session, and in the forth action, it receives an mrply message in the last phase in the current session. The server has three variables similar to those in the client. The server process can be defined as follows.

```
process server

const m, n  :   integer
                { m and n are as defined in process client}

var   phs   :   0..m,
      seq   :   0..n,
      b     :   boolean

begin
      rcv msg(b) from client -->
            if ~b --> skip
            []  b --> phs, seq := phs + 1, 0
            fi
```

```
[]      ODD.phs -->
            if seq < n-1    -->     send msg(false) to client;
                                    seq := seq + 1
            [] true         -->     send msg(true) to client;
                                    phs := phs + 1
            fi

[]      rcv mrqst from client -->
            {commit work of current session}
            send mrply to client; phs := 0
end
```

In this definition of a client/server protocol, we have abstracted away the lower protocol layers (for example socket protocols, TCP, IP, etc.) which transmit the messages between the client and the server and ensure that each transmitted message arrives at its destination in the same shape and same order in which it was sent. Instead of the lower protocol layers, we assumed that there are two direct channels between the client and the server, and that each message sent over these channels are guaranteed to be received in the same shape and same order in which it was sent. In other words, the two channels are perfect.

Correctness of the above client/server protocol depends strongly on the assumption that the channels between the client and server are perfect. Although this assumption is valid most of the time, it is possible that that some faults can cause this assumption to be invalid for a short time period, and when this happens for a client/server protocol, the protocol may reach and stay within unsafe states indefinitely. To overcome this possibility, we need to make the above client/server protocol stabilizing so that if it reaches (due to some fault) an unsafe state, the protocol is guaranteed to eventually return to safe states.

The method for making this client/server protocol stabilizing consists of three steps. First, assign a unique identifier to each session, and attach to each sent message in a session the unique identifier of that session. Second, modify both the client and server processes such that if any process receives a message whose attached session identifier is different from the identifier of the current session, the process aborts the current session. Third, modify the client process such that it waits long enough after it aborts a session before it starts the next session. These steps are discussed in more detail in the next three sections.

3 SESSION IDENTIFICATION

Each session is uniquely identified by a pair (snc, sns), where snc is the session identifier computed by the client and sns is the session identifier computed by the server. The client computes snc based on its local clock, and the server computes sns based on its local clock. The two clocks need not be synchronized in any way; the only requirement on each clock is that its successive readings are monotonically increasing. Thus, the two identifiers snc and sns of a session need not be related in any way.

The first two phases in each session are devoted to computing the identifier (snc, sns) of that session as follows. In the first phase, the client computes snc as the current value of its clock, then sends an srqst(snc) message to the server requesting the start of a new session. When the server receives the srqst(snc) message, it executes the second phase by computing sns as the current value of its clock then sending an srply(snc, sns) message to the client.

After the identifier (snc, sns) of a session is computed, it is attached to each message sent in that session. Thus, the messages sent in a session are of the following five forms: srqst(snc), srply(snc, sns), msg(snc, sns, b), mrqst(snc, sns), and mrply(snc, sns).

The new client process is similar to the client process in Section 2 with three exceptions. First, four integer variables snc, sns, c, and s are added to the client. Second, two actions are added to the client: one action for sending an srqst(snc) message and the other for receiving an srply(snc, sns) message. Third, a session identifier is attached to each sent or received message. The new client process is defined as follows.

```
process  client

const  m, n          :    integer

var    snc, c        :    integer,
       sns, s        :    integer,
       phs           :    0..m,
       seq           :    0..n,
       b             :    boolean

begin
       phs = 0  -->
              snc, phs := clock, 1; send srqst(snc) to server

[]     rcv srply(c, s) from server  -->
              sns, phs, seq := s, 2, 0
```

```
[]    EVEN.phs ^ 0 < phs < m-1 -->
          if seq < n-1   -->   send msg(snc, sns, false) to server;
                               seq := seq + 1
          [] true        -->   send msg(sns, s, true) to server;
                               phs := phs + 1
          fi

[]    rcv msg(c, s, b) from server -->
          if ~b --> skip
          []  b --> phs, seq := phs + 1, 0
          fi

[]    EVEN.phs ^ 0 < phs ≤ m-1 -->
          send mrqst(snc, sns) to server; phs := m

[]    rcv mrply(c, s) from server -->
          {work of current session is committed}
          phs := 0
end
```

Modifications similar to those in the client process are made to the server process. The new server process is defined as follows.

```
process server

const m, n   :    integer

var   snc, c :    integer,
      sns, s :    integer,
      phs    :    0..m,
      seq    :    0..n,
      b      :    boolean
begin
      rcv srqst(c) from client -->
          snc, sns, phs := c, clock, 2; send mrply(snc, sns) to client

[]    rcv msg(c, s, b) from client -->
          if ~b --> skip
```

```
                [] b --> phs, seq := phs + 1, 0
                fi
        [] ODD.phs -->
                if seq < n-1 --> send msg(snc, sns, false) to client;
                                 seq := seq + 1
                [] true      --> send msg(snc, sns, true) to client;
                                 phs := phs + 1
                fi
        [] rcv mrqst(c, s) from client -->
                {commit work of current session}
                send mrply(c, s) to client; phs := 0
        end
```

4 SESSION ABORTION

In the last section, we discussed how to identify each session and how to attach to each sent message the identifier of the session in which the message is sent. When a process, whether the client or the server, receives a message, it compares the identifier attached to the message with the identifier of the current session. If the two identifiers are equal, the process handles the message in the usual way. Otherwise, the process recognizes that a fault has occurred and aborts the current session as follows. The client aborts the current session by assigning a new value m+1 to its phs variable. The server aborts the current session and waits for the start of the next session by assigning its phs variable the value 0.

There are other situations where a process, whether the client or the server, detects a fault occurrence. If a process receives a message when its phs variable indicates that it should not be receiving this message, the process detects a fault occurrence and aborts the current session as discussed above.

In particular, the client aborts the current session when one of the following three events occurs. First, the client receives an srply(snc, sns) message when the value of its phs variable is not 1. Second, the client receives a msg(snc, sns) message when the value of its phs variable is even. Third, the client receives an mrply(snc, sns) message when the value of its phs variable is not m.

Similarly, the server aborts the current session when one of the following three events occurs. First, the server receives an srqst(snc) message when the value of its phs variable is not

0. Second, the server receives a msg(snc, sns) message when the value of its phs variable is odd or equals 0 or equals m-1. Third, the server receives an mrqst(snc, sns) message when the value of its phs variable is odd or equals 0.

The resulting client and server processes are defined as follows.

```
process client

const m, n          :   integer

var   snc, c        :   integer,
      sns, s        :   integer,
      phs           :   0..m+1,
      seq           :   0..n,
      b             :   boolean

begin
      phs = 0  -->
            snc, phs := clock, 1; send srqst(snc) to server

[]    rcv srply(c, s) from server  -->
            if snc ≠ c  ∨  phs ≠ 1 -->     phs := m+1
            [] snc = c  ^  phs = 1 -->     sns, phs, seq := s, 2, 0
            fi

[]    EVEN.phs  ^  0 < phs < m-1  -->
            if seq < n-1    -->    send msg(snc, sns, false) to server;
                                   seq := seq + 1
            [] true         -->    send msg(sns, s, true) to server;
                                   phs := phs + 1
            fi

[]    rcv msg(c, s, b) from server  -->
            if (snc ≠ c  ∨  sns ≠ s  ∨  EVEN.phs)                --> phs := m+1
            [] (snc = c  ^  sns = s  ^  ~EVEN.phs)  ^  ~b        --> skip
            [] (snc = c  ^  sns = s  ^  ~EVEN.phs)  ^   b        -->
                                                  phs, seq := phs + 1, 0
            fi
```

```
[]    EVEN.phs ^ 0 < phs ≤ m-1 -->
          send mrqst(snc, sns) to server; phs := m

[]    rcv mrply(c, s) from server -->
          if snc ≠ c ∨ sns ≠ s ∨ phs ≠ m --> phs := m+1
          [] snc = c ^ sns = s ^ phs = m --> phs := 0
          fi
end

process server

const m, n  :   integer

var   snc, c  :   integer,
      sns, s  :   integer,
      phs     :   0..m,
      seq     :   0..n,
      b       :   boolean
begin
      rcv srqst(c) from client -->
          if phs ≠ 0 --> phs := 0
          [] phs = 0 -->
                snc, sns, phs := c, clock, 2; send mrply(snc, sns) to client
          fi

[]    rcv msg(c, s, b) from client -->
          if (snc ≠ c ∨ sns ≠ s ∨ ODD.phs ∨ phs = 0 ∨ phs = m-1)
                                  -->   phs := 0
          [] (snc = c ^ sns = s ^ ~ODD.phs ^ phs ≠ 0 ^ phs ≠ m-1) ^
                              ~b  -->   skip
          [] (snc = c ^ sns = s ^ ~ODD.phs ^ phs ≠ 0 ^ phs ≠ m-1) ^
                               b  -->   phs, seq := phs + 1, 0
          fi

[]    ODD.phs -->
          if seq < n-1   -->   send msg(snc, sns, false) to client;
                               seq := seq + 1
          [] true        -->   send msg(snc, sns, true) to client;
```

```
                    phs := phs + 1
            fi
[]      rcv mrqst(c, s) from client -->
            if snc ≠ c ∨ sns ≠ s ∨ ODD.phs ∨ phs = 0 --> phs := 0
            [] snc = c ∧ sns = s ∧ ~ODD.phs ∧ phs ≠ 0 -->
                {commit work of current session}
                send mrply(c, s) to client; phs := 0
            fi
end
```

5 SESSION RESTART

Consider a state of the above protocol where the value of variable phs in the client is odd or equals m+1, the value of variable phs in the server is even, and the two channels between the client and the server are empty. This is a deadlock state because no action in the client or the server is enabled for execution at such a state. If due to some fault occurrence, the protocol reaches such a state, the execution of the protocol comes to a halt and no further progress by the client or the server is possible. To overcome this possibility, the following timeout action is added to the client.

```
timeout
        (In the client:  ODD.phs ∧ phs = m+1)      ∧
        (In the server:  EVEN.phs)                 ∧
        (The two channels between the client and server are empty)
                -->          phs := 0
```

In this action, variable phs in the client is assigned the value 0.

The guard of this action consists of three conjuncts. Only the first conjunct can be detected directly by the client. The other two conjuncts can be detected by the client indirectly. In particular, if the client waits for a long time, when the value of its variable phs is odd or equals m+1, without receiving any message from the server, then the client can deduce correctly that the current value of variable phs in the server is even and the two channels between the client and the server are currently empty.

6 PROOF OF STABILIZATION

In this section, we give a proof sketch of the stabilization property of the client/server protocol

in Sections 4 and 5. In particular, we argue that if this protocol starts executing at any state, then the protocol eventually reaches a state where the value of variable phs in the client is 0 and the two channels between the client and server are empty. (It is straightforward to show that any execution of the protocol starting from this state is "safe".)

To prove the stabilization property of this protocol, we need to discuss three other properties of the protocol; we refer to these properties as properties i to iii. As shown below, property i is needed for proving property ii, property ii is needed for proving property iii, and property iii is needed for proving the stabilization property of the protocol. These three properties are as follows.

i. Each execution of the protocol is infinite.

ii. In each (infinite) execution of the protocol, the client sends infinitely many messages.

iii. In each (infinite) execution of the protocol, the client executes its restart action, where it sends an srqst message, infinitely many times.

To prove property i, let s be an arbitrary state of the protocol. For s to be a deadlock state, then neither the client nor the server can send or receive messages at state s. Hence, the following predicate holds at s:

(In the client: ODD.phs / phs = m+1) ^
(In the server: EVEN.phs) ^
(The two channels between the client and server are empty)

However, this predicate is the guard of the timeout action in the client. In other words, the timeout action is enabled for execution at s, and s is not a deadlock state.

To prove property ii, note that executing each action of the server involves sending a message to the client or receiving a message from the client. Note also that the server cannot send more than n messages to the client without receiving a message from the client. From these facts and from property i, we conclude that the client sends infinitely many messages to the server in each execution of the protocol.

To prove property iii, note that the client cannot send more than n messages without incrementing its variable phs by at least one. Because the client sends infinitely many messages in each execution of the protocol, from property ii, and because variable phs in the client has an upper value of m+1, variable phs in the client is decremented infinitely many times in each execution of the protocol. Because variable phs in the client is decremented by being assigned the value 0, variable phs is assigned 0 infinitely many times in each execution of the protocol. One of two events occurs when the value of phs in the client is assigned 0: either the restart

action in the client is executed, or the client receives a message causing the value of phs to become m+1 and the client to execute its timeout action followed by its restart action. In either case, the restart action in the client is executed, after variable phs is assigned 0. Therefore, the restart action in the client is executed infinitely many times in each (infinite) execution of the protocol.

We now turn our attention to the stabilization property of the protocol. When the restart action in the client is executed, variable snc in the client is assigned the current value of the local clock, whose successive readings are monotonically increasing. From this fact and property iii, we conclude that the value of variable snc in the client is incremented infinitely many times in each (infinite) execution of the protocol. The protocol eventually reaches a state in which the value of variable phs in the client is 0 and the value of variable snc in the client is larger than or equal the value of variable snc in the server and larger than or equal the value of field snc in each message in the two channels between the client and the server. It is straightforward to argue that starting from this state, the protocol will reach a state where the value of variable phs in the client is 0 and the two channels between the client and the server are empty.

From this proof sketch, the stabilization property of the protocol does not depend on the sns variables or the sns message fields. These variables and fields are introduced solely to speed up the detection of fault occurrences in some cases.

Also from this proof sketch, the stabilization property of the protocol holds, if successive values of variable snc in the client are monotonically increasing. Thus, the protocol is still stabilizing, if the statement snc := **clock** in the client is replaced by the statement snc := snc + 1.

7 CONCLUDING REMARKS

In this paper, we identified a class of client/server protocols and discussed a method for making each protocol in this class stabilizing. Comparing the original protocol in Section 2 with the stabilizing protocol in Sections 4 and 5 should demonstrate that the cost of achieving stabilization is moderate. Given the high degree of fault tolerance that comes with stabilization, it seems that stabilization is a good bargain.

Dedication: This paper is dedicated to my daughter Nora on her seventeenth birthday.

8 REFERENCES

Arora, A. and Gouda M. G. (1993) Closure and convergence: A foundation of fault-tolerant Computing, IEEE Transactions on Software Engineering, Special Issue on Software Reliability, Vol. 19, No. 3, 1015-1027.

Gouda, M. G. (1995) The triumph and tribulation of system stabilization, Invited paper, Proceedings of the 9th International Workshop on Distributed Algorithms 1995 (WDAG'95), to be published by Springer-Verlag, Sept. 1995.

Gouda, M. G. and Multari, N. (1991) Stabilizing Communication Protocols, IEEE Transactions on Computers, Special Issue on Protocol Engineering, Vol. 40, No. 4, pp. 448-458.

9 BIOGRAPHY

Mohamed Gawdat Gouda studied Engineering (B. Sc. in 1968), and Mathematics (B. Sc. in 1971 and M. A. in 1972), before settling down on Computing Science (M. Math. 1973 and Ph. D. in 1977). He lived in Egypt (where he was born), and Canada (where he attended graduate schools), before ending up in Texas (where he earns his living). He enjoys working in industry (Honeywell, Bell Labs, and MCC), but seems content of his role as a Professor (The University of Texas at Austin). He works in a number of areas in computing science: formal methods, program verification and design, fault-tolerant computing, system stabilization, and network protocols, but the main objective of his work is always intellectual excitement and technical beauty.