

# Hello Again: Stabilization of the Hello Protocol in OSPF\*

Jorge A. Cobb <sup>†</sup>    Mohamed G. Gouda <sup>‡</sup>    Deepinder Sidhu <sup>§</sup>

## Abstract

The Hello protocol in OSPF allows each router in a network to check whether it can exchange messages with neighboring routers in its network. This check is carried out by making each router periodically send hello messages to every neighboring router in the network. Associated with the Hello protocol are two time periods: the hello period and the dead period. The hello period is the time period between sending two successive hello messages to the same neighbor. The dead period is the time period after which a router can declare a neighbor dead if during this period the router does not receive any hello messages from that neighbor. The original Hello protocol restricts the hello and dead periods to be fixed over time and to be identical in all routers. Simulation studies have shown that these restrictions contribute to network instabilities and even to network collapse. To improve network stability, we present a flexible Hello protocol where the hello and dead periods change over time and become consistent (rather than identical) in all routers. To ensure the fault-tolerance of our Hello protocol, the protocol is designed to be stabilizing. That is, when started from an arbitrary initial state, the protocol converges to a legitimate state, and remains in legitimate states throughout the remainder of its execution.

---

\*This work is supported by DARPA under contract F30602-00-C-0198

<sup>†</sup>Department of Computer Science (EC 31), The University of Texas at Dallas, Richardson, TX, 75083-0688, cobb@utdallas.edu.

<sup>‡</sup>Department of Computer Science, The University of Texas at Austin, 1 University Station C0500, Austin, TX 78712-0233, gouda@cs.utexas.edu.

<sup>§</sup>Telenix Corporation, 9194 Red Branch Road, Suite K, Columbia, MD 21045, dsidhu@telenix.com.

## 1 Introduction

A fundamental component of any computer network is its routing protocol. The purpose of the routing protocol is to direct each message along the most efficient path from its source computer to its destination computer. In order to determine the most efficient path, the routing protocol requires knowledge of the network topology.

In general, routing protocols learn the network topology in two steps. In the first step, each router obtains a list of its neighboring routers, that is, a list of those routers that are reachable via a single network link. In the second step, each router broadcasts this list to all other routers in the network. This broadcast may be explicit or implicit. In an explicit broadcast, the list of neighbors is broadcast without modification. This is known as link-state routing [14]. Examples of link-state routing protocols may be found in [4, 8, 13]. On the other hand, in an implicit broadcast, only distance-vectors are exchanged between neighbors. Multiple examples of distance-vector protocols may be found in the literature [5, 6, 7].

The above implies that an accurate list of neighboring routers is fundamental to a routing protocol. This list is obtained by exchanging messages, known as hello messages, between neighboring routers. If this exchange is possible, then the network link between the neighboring routers is functioning properly, and this link may be used to transfer data messages. The protocol in charge of the exchange of hello messages is known as the Hello protocol.

In the Internet, Open Shortest-Path First (OSPF) [4] is the most widely used routing protocol. It is a complex protocol, with multiple components. Its fundamental component, as argued above, is the Hello protocol.

There are two time periods associated with the Hello protocol in OSPF: the hello period and the dead period. The hello period is the time period between sending two successive hello messages to the same neighbor. The dead period is the time period after which a router can declare a neighbor dead if during this period the router does not receive any hello messages from that neighbor. The hello and dead periods have the following restrictions. For every pair of neighboring routers, the hello periods of these routers with respect to each other must be identical and fixed over time. Similarly, the dead periods of these two routers with respect to each other must be identical and fixed over time.

These restrictions have several shortcomings. In the first place, if the hello and dead periods are misconfigured at a router, then the router is unable to communicate with other routers, and is, in effect, removed from

the network. More importantly, simulation studies have shown that these restrictions contribute to network instabilities and even to network collapse [11, 12]. These same instabilities have been observed during the collapse of Internet service provider networks [11, 12].

The network instability begins when a major network event causes the broadcast of a large number of link-state advertisements and other control messages. This traffic of control messages causes hello messages to be excessively delayed or lost. In consequence, some routers incorrectly determine that a neighbor is dead. This in turn causes the broadcast of additional link-state advertisement and control messages, creating a cycle of instability.

Several techniques to mitigate this instability are recommended [11, 12]. For the Hello protocol, it is recommended that the hello and dead periods be adjusted in response to network overload. In this light, we present a Hello protocol with the following two properties. First, each router may vary its hello and dead periods as the need arises. As the hello and dead periods change, the state of the router (dead/live) is correctly determined by its neighbors. Second, timer management is very simple. In particular, each router maintains a single timer.

To ensure the fault-tolerance of our Hello protocol, the protocol is designed to be stabilizing. A protocol is said to be stabilizing if and only if, when started from an arbitrary initial state, it converges to a state contained in a set of legitimate states, and the set of legitimate states is closed under execution [9, 10]. Stabilizing protocols are robust, since they tolerate all forms of transient faults. That is, after a sequence of faults, a stabilizing protocol may be left in an illegitimate state, but within finite time, the protocol converges to a legitimate state, and remains within the set of legitimate states.

Our protocol is presented in four steps. We first present a Hello protocol as described in OSPF [4]. Next, we present a protocol where each router chooses its initial hello period, which, although fixed over time, need not be equal to the hello period of its neighboring routers. In addition, the router adapts its dead period according to two factors: the hello period chosen by its neighbor, and variations in network conditions. Then, we present a protocol where each router may vary its hello period in addition to varying its dead period. Finally, the protocol is strengthened to become stabilizing.

## 2 Protocol Notation and Network Model

Before presenting our protocols, we give a short overview of our protocol notation and network model. The notation is similar to that of Gouda [1].

A protocol consists of a set of  $n$  processes,  $p[0], p[1], \dots, p[n-1]$ , interconnected via communication channels. Each channel stores a sequence of messages between two processes. For every pair of processes  $p[i]$  and  $p[g]$ , if there is a channel from  $p[i]$  to  $p[g]$ , then there is also a channel from  $p[g]$  to  $p[i]$ . Two processes are said to be neighbors if and only if they are connected by a pair of channels.

The channel from process  $p[i]$  to process  $p[g]$  represents the local area network between the computers where  $p[i]$  and  $p[g]$  reside, plus any additional buffering at the computer of  $p[g]$  before the messages from  $p[i]$  are delivered to  $p[g]$ . Because communication between computers is direct, we assume channels do not reorder messages. In addition, we assume each channel can store a message for at most  $\lambda$  time units before the message is either received or lost.

Each process is specified by a set of inputs, a set of variables, a parameter, and a set of actions. The general structure of a process definition is as follows.

```
process <process name>
inp   <inp name> : <type>,
       <inp name> : <type>,
       . . .
       <inp name> : <type>
var   <var name> : <type>,
       <var name> : <type>,
       . . .
       < var name> : < type>
par   < par name> : < type>
begin
  < action>
[]
  < action>
[]
  . . .
[]
  < action>
```

**end**

The inputs declared in a process can be read, but not written, by the actions of that process. The variables declared in a process can be read and written by the actions of that process. The parameter is discussed below.

Every action is of the form:  $\langle \text{guard} \rangle \rightarrow \langle \text{command} \rangle$ . Executing an action consists of executing the statements in the command of the action. A command is constructed from sequencing (;) and iterative (**for**) constructs that group together assignment and send statements.

Assignment statements are of the form

$$\langle \text{variable} \rangle := \langle \text{expression} \rangle \text{ if } \langle \text{boolean expression} \rangle$$

If the  $\langle \text{boolean expression} \rangle$  is true before the statement is executed, then the  $\langle \text{variable} \rangle$  is assigned the current value of the  $\langle \text{expression} \rangle$ . If the  $\langle \text{boolean expression} \rangle$  is false, then the  $\langle \text{variable} \rangle$  remains unchanged. If the phrase **if**  $\langle \text{boolean expression} \rangle$  is not present, then the value of  $\langle \text{expression} \rangle$  is assigned to the  $\langle \text{variable} \rangle$  unconditionally.

A send statement in process  $p[i]$  is of the form **send msg to**  $p[g]$ , where  $p[g]$  is a neighbor of  $p[i]$ . Executing this statement appends a message of type  $msg$  to the channel from  $p[i]$  to  $p[g]$ .

The guard of an action is in one of three forms: local, receiving, and time-out. A local guard is a boolean expression over the inputs, variables, and parameter declared in the process. A local guard is enabled if and only if its boolean expression evaluates to true. A receiving guard is of the form **rcv msg from**  $p[g]$ , where  $p[g]$  is a neighbor of the process. This guard is enabled if and only if the next message to be received from neighbor  $p[g]$  is of type  $msg$ . Before the command of the receiving guard is executed, the received message is removed from the incoming channel. We describe time-out guards below.

Each process maintains multiple timer variables. Automatically, each timer variable is increased by one with each unit of time elapsed during the execution. A time-out guard is of the form **timeout**  $tr \geq t$ , where  $tr$  is the name of the timer, and  $t$  is an integer expression with the desired bound for the timer. This guard is enabled whenever the expression  $tr \geq t$  is true.

The parameter declared in a process is used to write a set of actions as a single action, with one action for each possible value of the parameter. For example, if we have the following parameter definition,

$$\text{par } g : 1 \dots 3$$

then the following action

$$\mathbf{rcv\ msg\ from\ } p[g] \rightarrow \mathbf{ack}[g] := \mathbf{true}$$

is a shorthand notation for the following three actions.

$$\begin{array}{l} \square \quad \mathbf{rcv\ msg\ from\ } p[1] \rightarrow \mathbf{ack}[1] := \mathbf{true} \\ \square \quad \mathbf{rcv\ msg\ from\ } p[2] \rightarrow \mathbf{ack}[2] := \mathbf{true} \\ \square \quad \mathbf{rcv\ msg\ from\ } p[3] \rightarrow \mathbf{ack}[3] := \mathbf{true} \end{array}$$

An execution step of the network consists in choosing, among all actions of all processes, an action whose guard is enabled, and executing this action. We assume that executing an action takes an arbitrary small amount of time. An execution of a protocol consists of a sequence of execution steps, which either never ends, or ends in a state where the guards of all actions are disabled.

Timer variables are incremented automatically as follows. Let  $s_i$  be the state of the protocol during the time interval  $[t_i, t_{i+1}]$ . Let action  $a_{i+1}$  be executed at time  $t_{i+1}$ , producing as a result the new state  $s_{i+1}$ . If a timer has the value  $x_i$  in state  $s_i$ , then the value of this timer when action  $a_{i+1}$  is chosen for execution is  $x_i + (t_{i+1} - t_i)$ .

There is an upper bound  $\Delta$  before an enabled action is selected for execution. That is, if an action is enabled at time  $t$ , then, by time  $t + \Delta$ , either the action is executed or it becomes disabled. We assume a similar, but tighter, upper bound  $\delta$  for time-out actions, where  $\delta \ll \Delta$ .

Finally, in a non-faulty initial protocol state, all channels are empty, and all timer variables have their maximum value. All other variables have their lowest value, i.e., integer variables have the value zero, and boolean variables have the value false.

### 3 The Original Hello Protocol

Consider a protocol of  $n$  processes  $p[0], p[1], \dots, p[n-1]$ , where each process represents a router in a network. Each process periodically sends a hello message to every neighboring process in the network. These hello messages allow each process  $p[i]$  to check, for each neighboring process  $p[g]$ , whether  $p[i]$  can exchange messages with  $p[g]$ .

Associated with the hello messages that a process  $p[i]$  periodically sends to a neighboring process  $p[g]$  is a time period called the *hello period* of  $p[i]$

with respect to  $p[g]$ : It is the time period that passes after  $p[i]$  sends a hello message to  $p[g]$  and before  $p[i]$  sends the next hello message to  $p[g]$ .

Associated with the hello messages that a process  $p[i]$  periodically receives from a neighboring process  $p[g]$  is a time period called the *dead period* of  $p[i]$  with respect to  $p[g]$ : If  $p[i]$  does not receive any hello message from  $p[g]$  during this period, then  $p[i]$  concludes that  $p[g]$  is dead at the end of this period.

Clearly, for any two neighboring processes  $p[i]$  and  $p[g]$ , the hello period of  $p[i]$  with respect to  $p[g]$  needs to be less than or equal to the dead period of  $p[g]$  with respect to  $p[i]$ .

In the original Hello protocol [4], the hello and dead periods have fixed values. Moreover, for any two neighboring processes  $p[i]$  and  $p[g]$ , the hello period of  $p[i]$  with respect to  $p[g]$  equals the hello period of  $p[g]$  with respect to  $p[i]$ , and the dead period of  $p[i]$  with respect to  $p[g]$  equals the dead period of  $p[g]$  with respect to  $p[i]$ . It follows that the hello period of  $p[i]$  with respect to  $p[g]$  is less than or equal the dead period of  $p[i]$  with respect to  $p[g]$ .

Each hello message received by a process  $p[i]$  from a neighboring process  $p[g]$  has three fields as follows:

$$\text{hello}(h, d, b)$$

where:  $h$  is the hello period of  $p[g]$  with respect to  $p[i]$ ,  
 $d$  is the dead period of  $p[g]$  with respect to  $p[i]$ , and  
 $b$  is a boolean bit with the following values:

$b = \mathbf{true}$  if  $p[g]$  has recently received hellos from  $p[i]$   
 $\mathbf{false}$  otherwise.

Each process  $p[i]$  in the network has the following three inputs:

**inp**  $N$  : **set**{  $g$  |  $p[g]$  is a neighbor of  $p[i]$ },  
 $hp$  : **array** [ $N$ ] **of**  $h_{min} .. h_{max}$ , / hello periods of  $p[i]$  /  
 $dp$  : **array** [ $N$ ] **of**  $d_{min} .. d_{max}$  / dead periods of  $p[i]$  /

Input  $N$  is the set of indices of all neighboring processes of  $p[i]$ . For every neighboring process  $p[g]$  of  $p[i]$ , the input  $hp[g]$  in  $p[i]$  is the hello period of  $p[i]$  with respect to  $p[g]$ , and the input  $dp[g]$  in  $p[i]$  is the dead period of  $p[i]$  with respect to  $p[g]$ .

The function of the Hello protocol is for each process  $p[i]$  to maintain a state variable  $st[g]$  for each neighboring process  $p[g]$ . The value of  $st[g]$ , in the range  $0 .. 2$ , is assigned according to the following rules:

- $st[g] = 0$  if  $p[i]$  is not receiving any hello messages from  $p[g]$ ,
- 1 if  $p[i]$  is receiving hello messages from  $p[g]$  but these messages indicate that  $p[g]$  is not receiving any hello messages from  $p[i]$ ,
  - 2 if  $p[i]$  is receiving hello messages from  $p[g]$  and these messages indicate that  $p[g]$  is also receiving hello messages from  $p[i]$ .

For each process  $p[i]$  to send a hello message to a neighboring process  $p[g]$  every  $hp[g]$  time units,  $p[i]$  maintains a timer  $tr[g]$ . Whenever this timer reaches the value of  $hp[g]$ ,  $p[i]$  sends a hello message to  $p[g]$ , and resets  $tr[g]$  to zero. Note that at all times  $tr[g]$  contains the amount of time elapsed since the last time  $p[i]$  sent a hello message to  $p[g]$ .

For each process  $p[i]$  to detect that its current dead period with respect to a neighboring process  $p[g]$  has expired,  $p[i]$  maintains a deadline variable  $dl[g]$  whose integer value is defined as follows.

- i. Each time  $p[i]$  times out to send a hello message to  $p[g]$ , the value of  $dl[g]$  is decremented by  $tr[g]$ , i.e., it is decremented by the amount of time elapsed from the previous time-out.
- ii. Anytime the value of  $dl[g]$  becomes zero,  $p[i]$  concludes that its current dead period with respect to  $p[g]$  has expired, and so it sets the value of  $st[g]$  to 0.
- iii. Each time  $p[i]$  receives a “proper” hello message from  $p[g]$ , the value of  $dl[g]$  is reset to  $dp[g] + tr[g]$ . Note that a hello message from  $p[g]$  to  $p[i]$  is “proper” if the hello period in the message equals the hello period of  $p[i]$  with respect to  $p[g]$  and the dead period in the message equals the dead period of  $p[i]$  with respect to  $p[g]$ .

A process  $p[i : 0 .. n - 1]$  in the Hello protocol can be specified as follows.

```

process  $p[i : 0 .. n - 1]$ 
inp    $N$        : set  $\{g \mid p[g] \text{ is a neighbor of } p[i]\}$ ,
           $hp$     : array  $[N]$  of  $h_{min} .. h_{max}$ ,           / hello periods of  $p[i]$  /
           $dp$     : array  $[N]$  of  $d_{min} .. d_{max}$            / dead periods of  $p[i]$  /
var    $tr$      : array  $[N]$  of  $0 .. h_{max} + \delta$ ,       / timers of  $p[i]$  /
           $st$     : array  $[N]$  of  $0 .. 2$ ,                   / states of neighbors of  $p[i]$  /
           $dl$     : array  $[N]$  of  $0 .. d_{max} + h_{max} + \delta$ , / deadlines of  $p[i]$  /
           $f$      :  $N$ ,                                       /  $p[f]$  is a neighbor of  $p[i]$  /
           $h$      :  $h_{min} .. h_{max}$ ,                           / fields in rcvd hello msg /

```

```

         $d$       :  $d_{min} \dots d_{max}$ ,
         $b$       : boolean
par    $g$       :  $N$  / any neighbor of  $p[i]$  /
begin
    timeout  $tr[g] \geq hp[g] \rightarrow$ 
         $dl[g] := dl[g] - tr[g];$ 
         $st[g] := 0$  if  $dl[g] = 0;$ 
        send  $hello(hp[g], dp[g], st[g] > 0)$  to  $p[g];$ 
         $tr[g] := 0$ 
    □
    rcv  $hello(h, d, b)$  from  $p[g] \rightarrow$ 
         $st[g] := 2$  if  $hp[g] = h \wedge dp[g] = d \wedge b;$ 
         $st[g] := 1$  if  $hp[g] = h \wedge dp[g] = d \wedge \neg b;$ 
         $st[g] := 0$  if  $hp[g] \neq h \vee dp[g] \neq d;$ 
         $dl[g] := dp[g] + tr[g]$ 
end

```

Each process has two actions. In the first action, process  $p[i]$  detects that its time-out action for a neighboring process  $p[g]$  is enabled for execution and executes it as follows. First,  $p[i]$  decrements the value of the deadline  $dl[g]$  by the amount of time elapsed, namely,  $tr[g]$ . If the value of  $dl[g]$  becomes zero, then  $p[i]$  determines that  $p[g]$  is unreachable, and assigns zero to  $st[g]$ . Second,  $p[i]$  sends a hello message to  $p[g]$ . Third,  $p[i]$  resets timer  $tr[g]$  to zero, and thus schedules its time-out action for  $p[g]$  to be executed once more after  $hp[g]$  time units.

In the second action, process  $p[i]$  receives a hello message from a neighboring process  $p[g]$  and updates its corresponding state  $st[g]$  and deadline  $dl[g]$  variables accordingly.

This protocol has two restrictions. First, the hello and dead periods of all processes have fixed values. Second, the hello messages received by any process  $p[i]$  from a neighboring process  $p[g]$  are accepted (by  $p[i]$ ) only if the hello and dead periods of  $p[i]$  with respect to  $p[g]$  are equal to, respectively, the hello and dead periods of  $p[g]$  with respect to  $p[i]$ . These two restrictions are needlessly severe, and in the rest of this paper we describe two variations of this protocol where these restrictions are relaxed.

## 4 A Protocol with Converging Dead Periods

In this section, we describe a variation of the Hello protocol where each process  $p[i]$  chooses its hello periods as it wishes, then each neighboring process  $p[g]$  adjusts its dead period with respect to  $p[i]$  to be consistent with the corresponding hello period of  $p[i]$ . In other words, the dead period of a neighboring process  $p[g]$  with respect to  $p[i]$  eventually “converges” to a value that is consistent with the chosen hello period of  $p[i]$  with respect to  $p[g]$ .

Because the hello periods of each process are chosen by the process, we design each process to choose all its hello periods to be equal. This design simplifies the time-out structure of each process. In particular, a process no longer needs to time-out at different time instants to send hello messages to different neighboring processes. Rather, when a process executes a time-out, it sends hello messages to everyone of its neighboring processes.

We next turn our attention to the question of how a process  $p[i]$  adjusts its dead period  $dp[g]$  with respect to a neighboring process  $p[g]$  to make it consistent with the hello period  $h$  chosen by  $p[g]$ . To answer this question, we assume that  $p[i]$  has a positive integer  $rf[g]$ , such that  $dp[g] := rf[g] \cdot h$ .

The value of  $rf[g]$  is chosen depending on the expected reliability of the channel from process  $p[g]$  to process  $p[i]$ . On one hand, if this channel is reliable and very few messages from  $p[g]$  to  $p[i]$  are ever lost, then  $rf[g]$  has a small value, say 1, 2, or 3. On the other hand, if this channel is unreliable and many messages from  $p[g]$  to  $p[i]$  can be lost, then  $rf[g]$  has a large value, say 8, 9, or 10. We refer to  $rf[g]$  as the reliability factor of process  $p[i]$  with respect to its neighboring process  $p[g]$ .

Note that the reliability of the channel may be affected by many factors, such as the probability of message corruption and the current level of channel congestion. For example, if the channel is being flooded by messages other than those in the hello protocol, such as data messages from network applications or routing control messages, then the value of  $rf[g]$  must be large enough to reflect these conditions. Therefore, we allow  $rf[g]$  to change over time, and thus, the dead period of  $p[i]$  with respect to  $p[g]$  will vary accordingly.

A process  $p[i : 0 .. n - 1]$  in the Hello protocol with converging dead periods can be specified as follows.

```

process  $p[i : 0 .. n - 1]$ 
inp    $N$        : set  $\{g \mid p[g] \text{ is a neighbor of } p[i]\}$ 
         $hp$       :  $h_{min} .. h_{max}$  / hello period of  $p[i]$  /

```

```

var   tr       : 0 ..  $h_{max} + \delta$ ,           / timer of  $p[i]$  /
        st       : array [ $N$ ] of 0 .. 2,         / states of neighbors of  $p[i]$  /
        dp       : array [ $N$ ] of  $d_{min} .. d_{max}$ ,   / dead periods of  $p[i]$  /
        dl       : array [ $N$ ] of 0 ..  $d_{max} + h_{max} + \delta$ , / deadlines of dead periods of  $p[i]$  /
        rf       : array [ $N$ ] of 1 ..  $r_{max}$ ,       / reliability factors of  $p[i]$  /
        f         :  $N$ ,                               /  $p[f]$  is a neighbor of  $p[i]$  /
        d         :  $d_{min} .. d_{max}$ ,                 / temporary variable /
        h         :  $h_{min} .. h_{max}$ ,                 / fields in rcvd hello msg /
        b         : boolean
par   g         :  $N$                                / any neighbor of  $p[i]$  /
begin
  timeout  $tr \geq hp \rightarrow$ 
    for every  $f$  in  $N$  do
       $dl[f] := dl[f] - tr$ ;
       $st[f] := 0$  if  $dl[f] = 0$ ;
      send  $hello(hp, st[f] > 0)$  to  $p[f]$ 
    od;
     $tr := 0$ 
  □
  rcv  $hello(h, b)$  from  $p[g] \rightarrow$ 
     $dp[g] := rf[g] \cdot h$ ;
     $dl[g] := dp[g] + tr$ ;
     $st[g] := 2$  if  $b$ ;
     $st[g] := 1$  if  $\neg b$ 
  □
  true  $\rightarrow$ 
     $h := dp[g]/rf[g]$ ;
     $rf[g] := \mathbf{any}$ ;
     $d := rf[g] \cdot h$ ;
     $dl[g] := dl[g] + (d - dp[g])$ ;
     $dp[g] := d$ 
end

```

This process  $p[i]$  is similar to process  $p[i]$  in Section 3, but there are four significant differences between the two.

- i. In this  $p[i]$ ,  $hp$  is a single input of type integer and  $dp$  is a variable array of integers. The value of  $hp$  is chosen before  $p[i]$  starts executing, and the value of each  $dp[g]$  is adjusted when  $p[i]$  receives a

hello message from its neighboring process  $p[g]$ . In  $p[i]$  in Section 3, both  $hp$  and  $dp$  are input arrays of integers, and their values are set before  $p[i]$  starts executing.

- ii. In this  $p[i]$ , there is exactly one time-out action, and when this time-out action is executed,  $p[i]$  sends a hello message to each of its neighboring processes. In  $p[i]$  in Section 3, there are several time-out actions: one action for each neighboring process of  $p[i]$ .
- iii. In this  $p[i]$ , every received hello message is accepted, and  $st[g]$  is updated according to the received boolean bit  $b$ . In  $p[i]$  in Section 3, a received  $hello(h, d, b)$  is accepted only if  $hp[g]$  in  $p[i]$  equals  $h$  and  $dp[g]$  in  $p[i]$  equals  $d$ .
- iv. For each neighbor  $p[g]$ , an additional action changes the value of  $rf[g]$  over time. The action first computes the current hello period of neighbor  $p[g]$ , and then chooses a new value for  $rf[g]$ . Then, the deadline and dead period with respect to  $p[g]$  are updated accordingly.

## 5 A Protocol with Dynamic Hello Periods

In the last Hello protocol, each process chooses its hello period when it starts executing, then it keeps its hello period fixed and lets its neighboring processes adjust their dead periods to become consistent with the chosen hello period. In this section, we discuss how to modify this protocol to allow any process to change its hello period when a need arises. For example, when a process becomes busy, it may increase its hello period in order to decrease the overhead that it encounters in sending hello messages. When the process is no longer busy, it may decrease its hello period to increase the responsiveness of the Hello protocol.

Assume that a process  $p[i]$  wishes to assign to its hello period  $hp$  a new value  $hn$ . There are two cases to consider. In the first case,  $hp \geq hn$ , and hence, the rate at which  $p[i]$  sends hello messages will rise. This higher message rate cannot increase the risk of a neighbor incorrectly declaring that  $p[i]$  is dead. Thus,  $p[i]$  may adopt the new hello period immediately by performing the assignment  $hp := hn$ .

In the second case,  $hp < hn$ , and thus, the rate at which  $p[i]$  sends hello messages will decrease. Since each neighbor has not yet increased its dead period with respect to  $p[i]$ ,  $p[i]$  cannot immediately adopt the larger hello

period without risking the possibility that one of these neighbors incorrectly declares that  $p[i]$  is dead. To prevent this possibility,  $p[i]$  keeps  $hp$  unchanged for sometime, but includes the new hello period  $hn$  in the hello messages that  $p[i]$  sends to its neighboring processes. Each neighboring process  $p[g]$  notices the new hello period  $hn$  in the hello messages (received from  $p[i]$ ), adjusts its dead period with respect to  $p[i]$  according to  $hn$ , and acknowledges the new hello period  $hn$  in the hello messages it sends to  $p[i]$ .

The acknowledgments are in the form of a small sequence number. Whenever  $p[i]$  chooses a new hello period  $hn$ , it increases its sequence number, and includes this sequence number in its hello messages. Each neighbor  $p[g]$  will acknowledge the new hello period by returning this sequence number to  $p[i]$  in its own hello messages. Therefore, each hello message from  $p[i]$  to  $p[g]$  contains two sequence numbers: the sequence number of the hello period at  $p[i]$ , and the sequence number of the hello period at  $p[g]$  that  $p[i]$  learned from the latest hello message it received from  $p[g]$ .

A process  $p[i : 0 \dots n-1]$  in the Hello protocol with dynamic hello periods can be specified as follows.

```

process  $p[i : 0 \dots n-1]$ 
inp    $N$        : set  $\{g \mid p[g] \text{ is a neighbor of } p[i]\}$ 
var    $tr$       :  $0 \dots h_{max} + \delta$ , / timer of  $p[i]$  /
         $st$       : array  $[N]$  of  $0 \dots 2$ , / states of neighbors of  $p[i]$  /
         $dp$       : array  $[N]$  of  $d_{min} \dots d_{max}$ , / dead periods of  $p[i]$  /
         $dl$       : array  $[N]$  of  $0 \dots d_{max} + h_{max} + \delta$ , / deadlines of dead periods of  $p[i]$  /
         $rf$       : array  $[N]$  of  $1 \dots r_{max}$ , / reliability factors of  $p[i]$  /
         $hp$       :  $h_{min} \dots h_{max}$ , / hello period of  $p[i]$  /
         $hn$       :  $h_{min} \dots h_{max}$ , / next hello period of  $p[i]$  /
         $ha$       : array  $[N]$  of boolean, / hello value is acknowledged /
         $sn$       :  $0 \dots s_{max} - 1$ , / sequence number of  $p[i]$  /
         $sg$       : array  $[N]$  of  $0 \dots s_{max} - 1$ , / sequence number of each neighbor /
         $f$        :  $N$ , /  $p[f]$  is a neighbor of  $p[i]$  /
         $d$        :  $d_{min} \dots d_{max}$ , / temporary variable /
         $h$        :  $h_{min} \dots h_{max}$ , / fields in rcvd hello msg /
         $s, s'$     :  $0 \dots s_{max} - 1$ ,
         $b$        : boolean
par    $g$        :  $N$  / any neighbor of  $p[i]$  /
begin
     $hp = hn \rightarrow$ 
         $hn := \mathbf{any}$ ;
         $hp := hn$  if  $hn \leq hp$ ;

```

```

     $sn := (sn + 1) \bmod s_{max}$  if  $hn > hp$ ;
     $ha[f] := \mathbf{false}$  if  $hn > hp$ 

```

□

```

timeout  $tr \geq hp \rightarrow$ 
    for every  $f$  in  $N$  do
         $dl[f] := dl[f] - tr$ ;
         $st[f] := 0$  if  $dl[f] = 0$ ;
        send  $hello(hn, sn, sg[f], st[f] > 0)$  to  $p[f]$ 
    od;
     $hp := hn$  if  $(\forall f : (st[f] = 2) \Rightarrow ha[f])$ ;
     $tr := 0$ 

```

□

```

rcv  $hello(h, s, s', b)$  from  $p[g] \rightarrow$ 
     $sg[g] := s$ ;
     $dp[g] := rf[g] \cdot h$ ;
     $dl[g] := dp[g] + tr$ ;
     $st[g] := 2$  if  $b$ ;
     $st[g] := 1$  if  $\neg b$ ;
     $ha[g] := (hs = s')$ 

```

□

```

true  $\rightarrow$ 
     $h := dp[g]/rf[g]$ ;
     $rf[g] := \mathbf{any}$ ;
     $d := rf[g] \cdot h$ ;
     $dl[g] := dl[g] + (d - dp[g])$ ;
     $dp[g] := d$ 

```

**end**

This process  $p[i]$  is similar to process  $p[i]$  in Section 4, but there are several significant additions.

- i. Four new variables are introduced: a new integer variable  $hn$ , a boolean array  $ha$ , a sequence number  $sn$ , and a sequence number array  $sg$ . In  $hn$ ,  $p[i]$  stores its new hello period. In  $ha[g]$ ,  $p[i]$  stores whether  $p[g]$  has acknowledged the new hello period. Finally,  $p[i]$  stores in  $sn$  the sequence number of its hello period, and stores in  $sg[g]$  the sequence number of the hello period received from  $p[g]$ .
- ii. A new action is introduced, where a new value is chosen for the

hello period. If the new hello period is smaller than the current hello period, then the new hello period is adopted immediately. Otherwise, all neighbors must be aware of the larger hello period before it may be adopted. Thus, the sequence number is increased, and all elements of array  $ha$  are set to false.

- iii. In the time-out action, the new hello period  $hn$  is assigned to the current hello period  $hp$ , provided that all neighbors with whom bidirectional communication has been established have acknowledged the new hello period.
- iv. In the receive action, the sequence number of neighbor  $p[g]$  is stored in  $sg[g]$ . In addition, if  $p[g]$  is acknowledging the current sequence number of  $p[i]$ , then  $ha[g]$  is set to true.

## 6 A Stabilizing Protocol

The protocol presented in the previous section is not strong enough to be stabilizing. For ease of presentation, we discuss in this section the necessary changes to ensure its stabilization, and present the complete protocol in the appendix, along with its proof of stabilization.

We begin by formally defining stabilization by borrowing some definitions from [2]. To simplify our stabilization predicate, we use the notion of pseudo-stabilization. We say a protocol *pseudo-stabilizes* to a predicate  $P$  in time  $T$  if and only if every execution of the protocol, regardless of its initial state, has an infinite suffix where  $P$  holds at every state in the suffix, and the execution reaches this suffix within  $T$  time units.

Below, whenever the process that declared a variable is not understood from context, the identity of the process is added as a suffix of the variable. E.g.,  $hp.i$  refers to variable  $hp$  in process  $p[i]$ .

Consider the following predicate.

$$(st[g].i = 2) \Rightarrow (dp[i].g \geq rf[i].g \cdot hp.i) \quad (1)$$

That is, if bidirectional communication has been established between  $p[i]$  and  $p[g]$ , then, the dead period of  $p[g]$  with respect to  $p[i]$  is large enough to accommodate the hello period of  $p[i]$ .

In order for the protocol of Section 5 to stabilize to (1), we must consider the contents of the channels between  $p[i]$  and  $p[g]$ . In particular, the channels at the initial state of the network may contain an arbitrary sequence of

messages. In this case, it is easy to obtain a race condition that prevents stabilization.

To ensure stabilization, we restrict how often each process changes its hello period. In particular, when a process increases its hello period, the outgoing and incoming channels of the process must be free of messages containing the new sequence number generated by the process. This is enforced by placing a lower bound  $\pi$  on the interval of time that may elapse between increases in the hello period of a process. We show in the appendix that  $\pi$  and the maximum sequence number,  $s_{max} - 1$ , must satisfy the following relationship.

$$s_{max} > \left\lfloor \frac{2 \cdot \lambda + d_{max} + h_{max} + \delta}{\pi} \right\rfloor + 1 \quad (2)$$

To implement the lower bound  $\pi$ , we introduce a new integer variable *inc*. This variable contains the remaining time units before *hn* may be increased once more. Variable *inc* is decremented by *tr* at each time-out. Process  $p[i]$  may choose a new value for *hn* only if *inc* = 0. Furthermore, if *hn* is increased, then *inc* is reset to  $\pi + tr$ .

Under a fault-free execution,  $hp \leq hn$  always holds. To ensure this holds in the presence of transient faults, the guard of the assignment statement  $hp := hn$  in the time-out action is weakened as follows.

$$hp := hn \quad \mathbf{if} \quad (\forall f : (hn > hp \wedge st[f] = 2) \Rightarrow ha[f])$$

In addition, process  $p[i]$  should not simply set  $st[g] = 2$  whenever the hello message from  $p[g]$  contains  $b = \mathbf{true}$ . This is because, even though  $b = \mathbf{true}$ ,  $p[g]$  may have considered  $p[i]$  to be dead temporarily, and during this time,  $p[i]$  changed sequence numbers. Therefore, the sequence number must also be taken into consideration. In particular, if  $st[g] = 1$ , then it should not be increased to two unless the right sequence number has already been received at  $p[g]$ . Also, if  $st[g] = 2$ , then it should be reduced to one if the wrong sequence number is received and  $p[i]$  is not changing its hello period (i.e.  $hn = hp$ ). Thus, the statement updating the value of  $st[g]$  in the receive action is modified as follows.

$$\begin{aligned} st[g] &:= 2 \quad \mathbf{if} \quad b \wedge hs = s'; \\ st[g] &:= 1 \quad \mathbf{if} \quad \neg b \vee (hn = hp \wedge hs \neq s'); \end{aligned}$$

Finally, to improve the convergence time, two new actions are introduced to ensure that  $dl[g] \leq dp[g] + tr$  and  $inc \leq \pi + tr$  hold at all times.

The above changes result in the stabilizing Hello protocol. The complete protocol is given in Appendix A. The following theorem is proven in Appendix B.

**Theorem 1** *The stabilizing Hello protocol pseudostabilizes to predicate (1) in time  $4 \cdot \lambda + 3 \cdot d_{max} + 3 \cdot h_{max} + \delta + \Delta$  irrespective of the number of message losses.*

## 7 Broadcast Channels

In the Internet, a computer network consists of a collection of subnetworks interconnected via routers. Two routers attached to a common subnetwork may exchange messages with each other via the common subnetwork. In general, there are two types of subnetworks: point-to-point and broadcast. To ensure fault-tolerance, the protocol is made stabilizing.

In a point-to-point subnetwork, the subnetwork is shared only by the routers at its two end points. This is represented in our notation by two processes connected by a pair of channels. Therefore, our Hello protocols are tailored for point-to-point subnetworks.

In a broadcast subnetwork, multiple routers are attached to the subnetwork. If any of these routers sends a message over the subnetwork, all other routers are able to receive a copy of this message. To represent a broadcast subnetwork, we connect processes via broadcast channels. When a process sends a message over a broadcast channel, all other processes connected to this channel receive a copy of the message.

Our Hello protocols can be tailored to efficiently operate over a broadcast channel as follows. When a process  $p[g]$  sends a hello message, it includes the following fields.

- Its own identifier  $g$ .
- Its hello period.
- Its hello sequence number.
- A list of identifiers. If identifier  $i$  is on this list, then  $st[i] \neq 0$  at  $p[g]$ , i.e.,  $p[g]$  received a hello message from  $p[i]$  before the dead period of  $p[g]$  with respect to  $p[i]$  expired.
- A list of sequence numbers, one for each identifier above. The sequence number associated with identifier  $i$  is the last sequence number that  $p[g]$  received from  $p[i]$ .

Given the above fields, when process  $p[i]$  receives a hello message from  $p[g]$ , it proceeds as follows. First,  $p[i]$  is aware that the message originates from  $p[g]$  because the identifier  $g$  is in the message. Second, if  $i$  is not listed in the hello message, then  $p[i]$  is aware that its hello messages are not reaching  $p[g]$ . On the other hand, if  $i$  is listed in the hello message, then  $p[i]$  can use the sequence number associated with identifier  $i$  to determine if  $p[g]$  has adjusted itself to the most recent hello period of  $p[i]$ .

## 8 Summary and Concluding Remarks

The original Hello protocol restricts the hello and dead periods to be fixed over time and be identical in neighboring routers. This, however, has been shown to contribute to network instabilities. In this light, we have presented two variations of the Hello protocol where the hello and dead periods change over time and become consistent, but not necessarily identical, in neighboring routers.

We have left open the question as to when should a router modify its hello period and how it should choose its new value. As indicated by Choudhury et. al. [12], automated methods can be used to detect that the network has reached an overloaded state. This would indicate that the hello period should be modified.

We have focused on the task of detecting if a neighboring router is reachable. However, the Hello protocol in OSPF has another important function in broadcast networks: electing a designated router and a backup designated router [4]. Each router indicates in its hello messages its “willingness” to become the designated router, and also indicates the pair of routers it currently considers to be the designated router and backup designated router. The standard algorithm to elect these routers is stabilizing without any additional modifications, and thus we do not include it in our protocols.

## References

- [1] Gouda, M., *Elements of Network Protocol Design*, Wiley-Interscience, 1998.
- [2] Gouda, M. G., Miller, R. E., Burns, J. E., “Stabilization and Pseudostabilization”, *Distributed Computing*, Vol. 7, No. 1., pp. 35-42, Nov. 1993.

- [3] Gouda M., “The Triumph and Tribulation of System Stabilization”, Proc. of the 9th International Workshop on Distributed Algorithms (WDAG) 1995, Lecture Notes in Computer Science 972.
- [4] Moy J., “OSPF Version 2”, Internet Request for Comments 2328, April 1998.
- [5] Hedrick, C., “Routing Information Protocol”, Internet Request for Comments 1058, 1988.
- [6] Hinden, R. and Sheltzer, A., “DARPA Internet Gateway Protocol, Internet Request for Comments RFC 823, September, 1982.
- [7] Garcia-Luna-Aceves, J. J., “Loop-Free Routing Using Diffusing Computations”, “IEEE/ACM Transactions on Networking”, Vol. 1, No. 1, February, 1993.
- [8] Garcia-Luna-Aceves, J. J. and Murthy S., “A Path-Finding Algorithm for Loop-Free Routing”, *IEEE/ACM Transactions on Networking*, Vol. 5, No. 1, February, 1997.
- [9] Schneider, M., “Self-Stabilization”, *ACM Computing Surveys*, Vol. 25, No. 1, 1993.
- [10] Herman, T., “A Comprehensive Bibliography on Self-Stabilization”, *Chicago Journal of Theoretical Computer Science*, working paper, 2002.
- [11] Ash J., Choudhury, G. L., Manral, V., Maunder, A., Sapozhnikova, V. D., Sherif, M., “Congestion Avoidance & Control for OSPF Networks”, Internet Draft (draft-ash-manral-ospf-congestion-control-00.txt), April, 2002
- [12] Ash, J., Choudhury, G. L., Han, J., Manral, V., Maunder, A., Noor-chashm, M., Sapozhnikova, V., D., Sherif, M., “Proposed Mechanisms for Congestion Control: Failure Recovery in OSPF & ISIS Networks”, Internet Draft (draft-ash-ospf-isis-congestion-control-02.txt), June, 2002
- [13] Vutukury, S., and Garcia-Luna-Aceves, J. J., “A Simple Approximation to Minimum Delay Routing”. *Proceedings of the SIGCOMM Conference*, 1999.
- [14] Tananbaum, A., *Computer Networks*, 4<sup>th</sup> ed., Prentice-Hall, 2003.

## A Stabilizing Protocol

```

process  $p[i : 0 \dots n - 1]$ 
inp    $N$        : set  $\{g \mid p[g] \text{ is a neighbor of } p[i]\}$ 
var    $tr$       :  $0 \dots h_{max} + \delta$ , / timer of  $p[i]$  /
         $st$       : array  $[N]$  of  $0 \dots 2$ , / states of neighbors of  $p[i]$  /
         $dp$       : array  $[N]$  of  $d_{min} \dots d_{max}$ , / dead periods of  $p[i]$  /
         $dl$       : array  $[N]$  of  $0 \dots d_{max} + h_{max} + \delta$ , / deadlines of dead periods of  $p[i]$  /
         $rf$       : array  $[N]$  of  $1 \dots r_{max}$ , / reliability factors of  $p[i]$  /
         $hp$       :  $h_{min} \dots h_{max}$ , / hello period of  $p[i]$  /
         $hn$       :  $h_{min} \dots h_{max}$ , / next hello period of  $p[i]$  /
         $ha$       : array  $[N]$  of boolean, / hello value is acknowledged /
         $sn$       :  $0 \dots s_{max} - 1$ , / sequence number of  $p[i]$  /
         $sg$       : array  $[N]$  of  $0 \dots s_{max} - 1$ , / sequence number of each neighbor /
         $inc$      :  $0 \dots \pi + h_{max} + \delta$ , / next time  $hn$  may increase /
         $f$        :  $N$ , /  $p[f]$  is a neighbor of  $p[i]$  /
         $d$        :  $d_{min} \dots d_{max}$ , / temporary variable /
         $h$        :  $h_{min} \dots h_{max}$ , / fields in rcvd hello msg /
         $s, s'$     :  $0 \dots s_{max} - 1$ ,
         $b$        : boolean
par    $g$        :  $N$  / any neighbor of  $p[i]$  /
begin
     $hp = hn \wedge inc = 0 \rightarrow$ 
         $hn := \mathbf{any}$ ;
         $hp := hn$  if  $hn \leq hp$ ;
         $sn := (sn + 1) \bmod s_{max}$  if  $hn > hp$ ;
         $ha[f] := \mathbf{false}$  if  $hn > hp$ ;
         $inc := \pi + tr$  if  $hn > hp$ 
□

    timeout  $tr \geq hp \rightarrow$ 
         $inc := inc - tr$ ;
        for every  $f$  in  $N$  do
             $dl[f] := dl[f] - tr$ ;
             $st[f] := 0$  if  $dl[f] = 0$ ;
            send  $hello(hn, sn, sg[f], st[f] > 0)$  to  $p[f]$ 
        od;
         $hp := hn$  if  $(\forall f : (hn > hp \wedge st[f] = 2) \Rightarrow ha[f])$ ;
         $tr := 0$ 

```

□

```

rcv hello(h, s, s', b) from p[g] →
    sg[g] := s;
    dp[g] := rf[g] · h;
    dl[g] := dp[g] + tr;
    st[g] := 2 if b ∧ hs = s';
    st[g] := 1 if ¬b ∨ (hn = hp ∧ hs ≠ s');
    ha[g] := (hs = s')

```

□

```

true →
    h := dp[g]/rf[g];
    rf[g] := any;
    d := rf[g] · h;
    dl[g] := dl[g] + (d - dp[g]);
    dp[g] := d

```

□

```

dl[g] > dp[g] + tr → dl[g] := dp[g] + tr

```

□

```

inc >  $\pi$  + tr → inc :=  $\pi$  + tr

```

**end**

## B Proof of Stabilization

**Lemma 1** *Let  $t$  be the initial (possibly faulty) state of the system.*

1. *From time  $t$  up to the first time-out,  $tr$  will increase by at least the amount of time between  $t$  and the time-out minus  $\delta$ .*
2. *At every subsequent time-out,  $tr$  contains the amount of time elapsed from the execution of the previous time-out to the execution of this time-out.*

*Proof:*

For the first part of the lemma, variable  $tr$  always increases with time, unless it has reached its highest value of  $h_{max} + \delta$ , at which point it stops increasing. However, at this time the time-out is enabled, and the time-out cannot remain enabled more than  $\delta$  time units without being executed.

For the second part of the lemma, at every time-out,  $tr$  is assigned zero. Furthermore, the time-out will be enabled no later than when  $tr$  reaches the value  $h_{max}$ . Since the largest value of  $tr$  has not been reached, and since the time-out will be executed within  $\delta$  time units,  $tr$  always increases between time-outs.

*End of Proof.*

**Lemma 2** *Let  $t$  be the time of the initial (possibly faulty) state of an execution. For all time  $t'$ , where  $t + \Delta \leq t'$ , the following hold.*

$$\begin{aligned} dl[g] &\leq dp[g] + tr \\ inc &\leq \pi + tr \end{aligned}$$

*Proof:*

After  $\Delta$  time units, the final two actions will execute. Furthermore, the value of  $tr$  increases with time, and when it is reset to zero it is first subtracted from  $dl[g]$  and  $inc$ , preserving the above relations. When  $dl[g]$  and  $inc$  are reset, their new values satisfy the above relations. Finally, when  $rf[g]$  changes,  $dl[g]$  and  $dp[g]$  are increased by the same amount.

*End of Proof:*

**Lemma 3** *Let  $t$  be the time of the initial (possibly faulty) state of an execution. During any interval of time  $[t', t'']$ , the maximum number of times a process can increase the value of  $hn$  is*

$$\left\lfloor \frac{t'' - t'}{\pi} \right\rfloor + 1$$

*Proof:*

In the worst case,  $hn$  is increased exactly at time  $t'$ . Then,  $inc$  is set to  $\pi + tr$ . After the first time-out, the new value of  $inc$  will be  $\pi$  minus at most the amount of time elapsed since  $hn$  increased. The reason it may not be equal to the elapsed time is because the first time-out may be late by  $\delta$  time units (Lemma 1). For every subsequent time-out,  $inc$  will decrement by the time elapsed from the previous time-out. Thus,  $inc$  will be set to zero, and  $hn$  may increase, no more often than once every  $\pi$  time units. If we count the initial increase at  $t$ , the number of times  $hn$  may increase is  $\left\lfloor \frac{t'' - t'}{\pi} \right\rfloor + 1$ .

*End of Proof.*

**Lemma 4** *Let  $t$  be the initial (possibly faulty) state of an execution, and let  $t + \Delta \leq t'$ . If from time  $t'$  up to time  $t' + d_{max} + h_{max} + \delta$  no message is received from a neighbor, then the neighbor has been declared dead by time  $t'$ .*

*Proof:*

We consider how much time may elapse until  $dl[g]$  is assigned zero. After time  $t'$ , Lemma 2 holds, i.e.,  $dl[g] \leq dp[g] + tr$ . Therefore, since  $dl[g]$  is decreased by  $tr$  at the time of the first time-out, and since  $tr$  may not increase during the execution for  $\delta$  time units (Lemma 1), then at the first time-out  $dl[g]$  is decreased by at least the time elapsed from  $t'$  to the time-out time minus  $\delta$ . After every subsequent time-out (Lemma 1),  $dl[g]$  is decreased by the duration of the time-out. Furthermore, since each time-out is at least  $hp$  time units from the previous one, and the maximum value of  $hp$  is  $h_{max}$ , the last time-out may be up to  $h_{max}$  time units greater than necessary. In addition, the initial value of  $dp[g]$  is at most  $d_{max}$ , and thus at time  $t'$  we have  $dl[g] \leq d_{max} + tr$ . Finally, the value of  $dl[g]$  may increase or decrease in the last action. However, its relative change of value is equal to that of  $dp[g]$ , and  $dp[g]$  is bounded by  $d_{max}$ . Hence, no later than time  $t' + d_{max} + h_{max} + \delta$  a time-out will execute setting  $dl[g]$  to zero and declaring  $p[g]$  unreachable.

*End of Proof*

We say that  $p[g]$  agrees with the  $hn$  value of  $p[i]$ , denoted  $hn\_agrees(g, i)$ , if all of the following hold.

1.  $dp[i].g \geq rf[i].g \cdot hn.i \wedge sg[i].g = hs.i$
2. For every message  $hello(h, s, s', b)$  in the channel from  $p[i]$  to  $p[g]$ ,

$$hn.i \leq h \wedge hs.i = s$$

3. For every message  $hello(h, s, s', b)$  in the channel from  $p[g]$  to  $p[i]$ ,

$$b \Rightarrow (hs.i = s')$$

It is easy to show that  $hn\_agrees(g, i)$  is stable under the execution of actions in  $p[g]$ .

**Lemma 5** *For any  $x$ , if  $dp[i].g \geq rf[i].g \cdot x$  holds, then it continues to hold after executing the fourth action of process  $p[g]$ .*

*Proof:*

This follows imply from algebra.

*End of Proof*

**Lemma 6** *If  $hn\_agrees(g, i)$  holds, then it continues to hold after the execution of any action in  $p[g]$ .*

*Proof:*

The first action of  $p[g]$  does not reference any value in  $hn\_agrees(g, i)$ . In the second action of  $p[g]$ , it only affects  $hn\_agrees(g, i)$  by sending a hello message to  $p[i]$ . However, since part 1 in  $hn\_agrees(g, i)$  holds, the message will have the correct sequence number. In the third action,  $dp[i].g$  and  $sg[i].g$  are updated according to the message received from  $p[i]$ , but due to part 2 of  $hn\_agrees(g, i)$ , the new values satisfy  $hn\_agrees(g, i)$ . In the fourth action,  $dp[i].g$  is modified, but from Lemma 5, the new value also satisfies  $hn\_agrees(g, i)$ . The fifth and sixth actions do not reference values in  $hn\_agrees(g, i)$ .

*End of Proof*

**Lemma 7** *Let  $t$  be the time of the initial (possibly faulty) state of the execution. For any time  $t'$ , where  $t + 2 \cdot \lambda + d_{max} + h_{max} + \delta + \Delta \leq t'$ , if there exists a message  $hello(h, s, s', b)$  at the head of the channel from  $p[g]$  to  $p[i]$ , such that  $b \wedge s' = sn.i$ , then  $hn\_agrees(g, i)$  holds.*

*Proof:*

We will call any message sent by  $p[i]$  to  $p[g]$  after time  $t + \Delta$  “fresh”. After time  $t + \Delta + \lambda$ , all messages from  $p[i]$  to  $p[g]$  have to be fresh, due to the upper bound  $\lambda$  on message lifetime. A message is called “live” if its last field, the boolean value, equals true.

We would like to obtain a lower time bound such that any live message sent by  $p[g]$  after this time bound must be preceded by  $p[g]$  receiving a fresh message from  $p[i]$ .

Recall that, after time  $t + \lambda + \Delta$ , only fresh messages are in the channel from  $p[i]$  to  $p[g]$ . If from time  $t + \lambda + \Delta$  to time  $t + \lambda + d_{max} + h_{max} + \delta + \Delta$  a fresh message is not received by  $p[g]$  from  $p[i]$ , then it implies that  $p[g]$  has declared  $p[i]$  dead for not receiving a hello message (Lemma 4). Thus, any live message transmitted by  $p[g]$  after time  $t + \lambda + d_{max} + h_{max} + \delta + \Delta$  is transmitted after the reception of a fresh message. Therefore, any live message in the channel from  $p[g]$  to  $p[i]$  at time  $t + 2 \cdot \lambda + d_{max} + h_{max} + \delta + \Delta$  or greater must be transmitted after the reception of a fresh message.

Let  $m$  be a live message that is at the head of the channel from  $p[g]$  to  $p[i]$  at time  $T$ , where  $T \geq t + 2 \cdot \lambda + d_{max} + h_{max} + \delta + \Delta$ . From above,  $p[g]$  must have received a fresh message from  $p[i]$  before transmitting  $m$ . Let  $m'$  be the last message  $p[g]$  received from  $p[i]$  before sending  $m$ . We want to consider a lower bound on the time at which  $m'$  was sent by  $p[i]$ .

Message  $m$  was sent no earlier than at time  $T - \lambda$ , since messages only live  $\lambda$  time units. Furthermore, since the message is live,  $p[g]$  must have received

message  $m'$  from  $p[i]$  no earlier than time  $T - \lambda - d_{max} - h_{max} - \delta$ , since otherwise  $p[g]$  would have timed out and considered  $p[i]$  to be nonreachable. Finally, message  $m'$  could not have been sent any earlier than  $T - 2 \cdot \lambda - d_{max} - h_{max} - \delta$  due to lifetime of messages in channels.

From above, a maximum of  $2 \cdot \lambda + d_{max} + h_{max} + \delta$  time units have elapsed from the transmission of  $m'$  until time  $T$ , i.e., until the time  $m$  is at the head of the channel from  $p[g]$  to  $p[i]$ . From relation (2), there is not enough time for the sequence numbers to wrap around. Hence,  $hn.i$  has not increased since the time  $p[i]$  sent  $m'$ .

We next consider the three components of  $hn\_agrees(g, i)$  and show that they hold at time  $T$ .

- Since  $hn.i$  has not increased nor the sequence number changed since the time  $p[i]$  sent  $m'$ , all messages sent by  $p[i]$  after  $m'$  satisfy part 2 of  $hn\_agrees(g, i)$ .
- From the definition of  $m'$ ,  $p[g]$  received  $m'$ , stored the sequence number of  $p[i]$ , and adjusted  $dp[i].g$  accordingly, and thus part 1 of  $hn\_agrees(g, i)$  held when  $m'$  was received. From part 2, this continues to hold whenever  $p[g]$  receives a message from  $p[i]$ . Also, from Lemma 5, part 1 continues to hold when  $p[g]$  updates  $rf[i]$ .
- Because part 1 holds after  $m'$  is received at  $p[g]$ , any message sent by  $p[g]$  after sending  $m$  satisfies part 3.

*End of Proof.*

**Lemma 8** *Let  $t$  be the time of the initial (possibly faulty) state of the execution. Then, at any time  $t'$ , where  $t + 2 \cdot \lambda + 2 \cdot d_{max} + 2 \cdot h_{max} + \delta + \Delta \leq t'$ , the following will hold and continue to hold for every process  $p[i]$  and its neighbor  $p[g]$ .*

$$(ha[g].i \wedge st[g].i = 2) \Rightarrow hn\_agrees(g, i) \quad (3)$$

*Proof:*

Let  $T = t + 2 \cdot \lambda + d_{max} + h_{max} + \delta + \Delta$ . We first argue that if predicate (3) holds at any time after  $T$ , then it will continue to hold. If predicate (3) holds, it may be falsified when the left-hand-side is false and an action makes it true, or when the right-hand-side is true and an action makes it false. Consider the first case. If the left-hand-side is false, it can only become true by receiving a hello message from  $p[g]$ . If the message has an incorrect sequence number,

$ha[g].i$  is set to false, and predicate (3) holds. If the message has the correct sequence number, then from Lemma 7,  $hn\_agrees(g, i)$  holds, and hence predicate (3) also holds. Consider the second case. If the right-hand-side is true, then from Lemma 6, no action of  $p[g]$  may falsify it. On the other hand,  $p[i]$  may falsify  $hn\_agrees(g, i)$  by increasing the value of  $hn$  (and at the same time changing the value of  $sn$ ), but in this case  $ha[g].i$  is set to false, and predicate (3) still holds.

To show that predicate (3) will hold after time  $T + d_{max} + h_{max}$ , assume that a hello message is received by  $p[i]$  from  $p[g]$  from time  $T$  to time  $T + d_{max} + h_{max}$ . If the left-hand-side of the predicate is true after the action, this implies the sequence number of the message is correct, in which case by Lemma 7 we have that  $hn\_agrees(g, i)$  holds, and predicate (3) holds. Assume no hello message is received during this interval, then, from Lemma 4<sup>1</sup>,  $p[i]$  will declare  $p[g]$  dead and assign zero to  $st[g].i$ , and predicate (3) holds.

*End of Proof*

**Lemma 9** *Let  $t$  be the time of the initial faulty state of the execution. At any time  $t'$ , where  $t + h_{max} + \delta \leq t'$ ,  $hp \leq hn$  holds at every process.*

*Proof:*

The time-out becomes enabled no later than time  $t + h_{max}$ , because the minimum value of  $tr$  is zero and the maximum value of  $hp$  is  $h_{max}$ . Thus, the time-out will be executed no later than time  $t + h_{max} + \delta$ . After the first time-out, if  $hp > hn$ , then  $hn$  is assigned to  $hp$ , ensuring  $hp \leq hn$ . Whenever  $hn$  changes in the first action, it must be that  $hp = hn$  before the action. Furthermore, if the new value of  $hn$  is less than that of  $hp$ , then  $hn$  is assigned to  $hp$  immediately. If  $hp < hn$ , then it remains so until in the time-out action  $hn$  is assigned to  $hp$ . The lemma thus follows.

*End of Proof*

We say that  $p[g]$  agrees with the  $hp$  value of  $p[i]$ , denoted  $hp\_agrees(g, i)$ , if the following two conditions hold.

- $(st[i].g > 0) \Rightarrow (dp[i].g \geq rf[i].g \cdot hp.i)$
- For every message  $hello(h, s, s', b)$  in the channel from  $p[i]$  to  $p[g]$ ,

$$hp.i \leq h$$

---

<sup>1</sup>Note that an additional term  $\delta$  is not necessary because, from the proof of Lemma 4, the term  $\delta$  originates from the inaccuracy of the time-out mechanism, which from Lemma 1 will occur only at the first time-out. From the value of  $T$ , at least one time-out must have occurred before  $T$ . Hence, not time-out inaccuracies will occur after  $T$ .

**Lemma 10** *If  $hp\_agrees(g, i)$  holds, then it continues to hold after the execution of any action in  $p[g]$ .*

*Proof:*

The first action of  $p[g]$  does not reference any value in  $hp\_agrees(g, i)$ .

In the second action of  $p[g]$ , it only affects  $hp\_agrees(g, i)$  by changing the value of  $st[i].g$ . However, the new value is zero, and hence,  $hp\_agrees(g, i)$  holds.

In the third action,  $dp[i].g$  and  $st[i].g$  are updated according to the message received from  $p[i]$ , but due to part 2 of  $hp\_agrees(g, i)$ , the new values satisfy  $hp\_agrees(g, i)$ .

In the fourth action,  $dp[i].g$  is modified, but from Lemma 5, the new value also satisfies  $hp\_agrees(g, i)$ .

The fifth and sixth actions do not reference values in  $hp\_agrees(g, i)$ .

*End of Proof*

**Theorem 2** *Let  $t$  be the time of the initial (possibly faulty) state of the execution. Then, at any time  $t'$ , where  $t + 3 \cdot \lambda + 2 \cdot d_{max} + 2 \cdot h_{max} + \delta + \Delta \leq t'$ , the following holds for every process  $p[i]$  and its neighbor  $p[g]$ .*

$$(st[g].i = 2) \Rightarrow hp\_agrees(g, i) \quad (4)$$

*Proof:*

Let  $T = t + 2 \cdot \lambda + d_{max} + h_{max} + \delta + \Delta$ . We first show that if after time  $T$  predicates (3) and (4) hold concurrently, then they continue to hold. From the proof of Lemma 8, if predicate (3) holds at any time after  $T$ , then it continues to hold. Thus, we focus on predicate (4).

Process  $p[g]$  cannot affect (4), because the left-hand-side of (4) only refers to variables in  $p[i]$ , and from Lemma 10,  $p[g]$  cannot falsify the right-hand-side.

Consider now process  $p[i]$ . In the first action, it may change the value of  $hn.i$  but not of  $hp.i$ , and thus, it cannot affect (4). In the time-out action, if  $st[g]$  is set to zero then (4) holds trivially. Otherwise, from Lemma 9,  $hp \leq hn$ , and hence, the new hello message sent agrees with  $hp$ . Furthermore, if  $hp$  is assigned  $hn$ , then from the guard of the assignment, either  $st[g] < 2$ , in which case (4) holds trivially, or we have  $ha[g] \wedge st[g] = 2$ . In the latter case, since (3) holds, then  $hn\_agrees(g, i)$  holds, and thus  $hp\_agrees(g, i)$  will hold after the assignment. In the receive action, (4) is only affected by assigning 2 to  $st[g]$ . In this case, however,  $b \wedge hs = s'$ , and from Lemma 7,  $hn\_agrees(g, i)$  holds, and from Lemma 9,  $hp\_agrees(g, i)$  will hold after the assignment. The last action does not affect (4).

We next need to show that predicate (4) will hold and continue to hold. We also show that (3) also holds at this time, and hence, both predicates will continue to hold. Below, we refer to time  $T + \lambda + d_{max} + h_{max}$  as  $T'$ . Note that from the proof of Lemma 8, if predicate (3) holds at any time after  $T$ , it continues to hold, and furthermore, it is guaranteed to hold and continue to hold no later than time  $T + d_{max} + h_{max}$ , i.e., earlier than  $T'$ .

Assume that at some time from  $T$  to  $T'$  we have  $st[g].i < 2$ . In this case (4) and (3) already hold, and thus they will continue to hold. We thus focus on the case where  $st[g].i = 2$  at time  $T$  and remains so until time  $T'$ .

Similar to the proof of Lemma 7, any message sent by  $p[i]$  to  $p[g]$  after time  $T$  is called “fresh”. After time  $T + \lambda$ , all messages from  $p[i]$  to  $p[g]$  have to be fresh, due to the upper bound  $\lambda$  on message lifetime.

Assume that  $hp.i$  is always smaller than the hello values of all the fresh messages sent out by  $p[i]$ , and this holds from time  $T$  to time  $T'$ . Then,  $hp\_agree(g, i)$  should hold. This reason is similar to the proof of Lemma 7. If by time  $T + \lambda + d_{max} + h_{max}$ , i.e.  $T'$ ,  $p[g]$  did not receive a fresh message from  $p[i]$ ,  $p[g]$  declares  $p[i]$  dead (Lemma 4 and Lemma 1), and  $hp\_agree(g, i)$  holds. Thus, (4) holds, and we know that (3) holds and continues to hold earlier than  $T'$ . If  $p[g]$  does receive a fresh message from  $p[i]$  within time  $T'$ , then its dead period with respect to  $p[i]$  is in agreement with  $hp.i$ , and (4) holds. Furthermore, all fresh messages sent from time  $T$  to time  $T'$  are at least  $hp.i$ , thus, the dead period of  $p[g]$  with respect to  $p[i]$  will remain in agreement with  $hp.i$  from  $T$  to  $T'$ , thus, (4) continues to hold, and lastly, (3) will hold and continue to hold at a time earlier than  $T'$ . Hence, both (3) and (4) will hold and continue to hold.

On the other hand, assume that, somewhere from time  $T$  to time  $T'$ ,  $hp.i$  becomes larger than the hello value of a fresh message from  $p[i]$  to  $p[g]$ . At time  $T$ , from Lemma 9,  $hp.i \leq hn.i$ . Thus, since hello values in messages are copied from  $hn.i$ ,  $hp.i$  will be at most the hello value of the first fresh messages. To become larger than one of them,  $hn.i$  must increase, and therefore, array  $ha.i$  is set to false, and later  $hp.i$  is assigned  $hn.i$ . However, since we assume that  $st[g].i = 2$ , for  $hp.i$  to be assigned  $hn.i$  it must be that  $ha[g].i$  changed from false to true. This implies receiving a message from  $p[g]$  with the appropriate sequence number, and from Lemma 7,  $hn\_agrees(g, i)$  holds, and from Lemma 9,  $hp\_agrees(g, i)$  also holds. Thus, both (4) and (3) will hold and continue to hold.

*End of Proof*

We say that  $p[g]$  fully agrees with the  $hp$  value of  $p[i]$ , denoted  $hp\_fully\_agrees(g, i)$ , if the following two conditions hold.

- $dp[i].g \geq rf[i].g \cdot hp.i$
- For every message  $hello(h, s, s', b)$  in the channel from  $p[i]$  to  $p[g]$ ,

$$hp.i \leq h$$

**Lemma 11** *If  $hp\_fully\_agrees(g, i)$  holds, then it continues to hold after the execution of any action in  $p[g]$ .*

*Proof:*

Neither the first nor second actions of  $p[g]$  reference any value in  $hp\_fully\_agrees(g, i)$ .

In the third action,  $dp[i].g$  is updated according to the message received from  $p[i]$ , but due to part 2 of  $hp\_fully\_agrees(g, i)$ , the new values satisfy  $hp\_fully\_agrees(g, i)$ .

In the fourth action,  $dp[i].g$  is modified, but from Lemma 5, the new value also satisfies  $hp\_fully\_agrees(g, i)$ .

The fifth and sixth actions do not reference values in  $hn\_agrees(g, i)$ .

*End of Proof*

**Theorem 3** *Let  $t$  be the time of the initial (possibly faulty) state of the execution. Then, at any time  $t'$ , where  $t + 4 \cdot \lambda + 3 \cdot d_{max} + 3 \cdot h_{max} + \delta + \Delta \leq t'$ , the following holds for every process  $p[i]$  and its neighbor  $p[g]$ .*

$$(st[g].i = 2) \Rightarrow hp\_fully\_agrees(g, i) \quad (5)$$

*Proof:*

Let  $T = t + 2 \cdot \lambda + d_{max} + h_{max} + \delta + \Delta$ ,  $T' = T + \lambda + d_{max} + h_{max}$ , and  $T'' = T' + \lambda + d_{max} + h_{max}$ . First, using Lemma 11, a proof similar to the first part of the proof of Theorem 2 shows that if predicate (3) and predicate (5) hold together at any time after time  $T$ , then they continue to hold. We are guaranteed from Lemma 8 that (3) holds after time  $T'$ . Hence, we focus on showing that (5) holds after time  $T'$  and before time  $T''$ . Also, from Theorem 2, predicate (4) holds by time  $T'$ .

If (5) holds at time  $T'$ , we are done. Otherwise, since (4) does hold, the only choice is for  $st[i].g = 0$ . If  $st[i].g$  becomes greater than zero, then from (4) we have that (5) holds and we are done. Otherwise, all messages sent by  $p[g]$  after  $T'$  are not live. If any of these non-live messages arrives to  $p[i]$  by time  $T''$ , then  $st[g].i < 2$  will hold, and (5) holds. Otherwise, by time  $T' + \lambda$ , any live message from  $p[g]$  to  $p[i]$  is gone and only non-live messages exist

from  $p[g]$  to  $p[i]$ . Thus, by time  $T' + \lambda + d_{max} + h_{max}$  (Lemma 4),  $p[i]$  will timeout and assign zero to  $st[g].i$ , and (5) holds.

*End of Proof*

Theorem 3 implies Theorem 1, and the proof of the stabilizing protocol is therefore complete.