Accelerated Heartbeat Protocols*†

Mohamed G. Gouda

gouda@cs.utexas.edu
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Tommy M. McGuire

mcguire@cs.utexas.edu
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Abstract

Heartbeat protocols are used by distributed programs to ensure that if a process in a program terminates or fails, then the remaining processes in the program terminate. We present a class of heartbeat protocols that tolerate message loss. In these protocols, a root process periodically sends a beat message to every other process then waits to receive a reply beat message from every other process. If the root process does not receive a reply (possibly due to message loss), the root process reduces by half the period for sending beat messages. We show that in practical situations, the parameters of these protocols can be chosen to achieve a good compromise between three contradictory objectives: reduce the rate of sending beat messages, reduce the detection delay, and still keep the probability of premature termination small.

1 Introduction

A fundamental construct for tolerating faults in computer networks is a heartbeat protocol. A heartbeat protocol allows processes in the same program in a network to periodically exchange *beat* messages. As long as a process p keeps receiving *beat* messages from a process q, process p recognizes that process q and the communication medium from q to p are both up. If p does not receive any *beat* messages from q for a long time, p recognizes that q has terminated or failed or that the communication medium from q to p has failed. In this case, process p itself terminates. Therefore, a heartbeat protocol ensures that if one or more processes in a program fail or terminate, then every other process in the program terminates. Many uses of heartbeat protocols are reported in the literature. For example, they are used in system diagnosis[1], network protocols[2], reaching agreement[4], mobile computing[9], and fault detection in computer networks[12].

Heartbeat protocols differ from termination detection algorithms (as discussed for example in [3, 5, 6]) in two important ways. First, a heartbeat protocol causes all processes to terminate when one or more processes terminate, whereas a termination detection algorithm merely detects that termination has occurred when all processes do terminate. Second, a heartbeat protocol causes all processes to terminate when the communication medium between the processes fails, whereas the correctness of a termination detection algorithm is usually based on the assumption that the communication medium does not fail or that its failure can be detected by a heartbeat protocol[11].

In designing a heartbeat protocol, the protocol designer strives to achieve the following three objectives.

1. The rate at which *beat* messages are sent in the protocol should be small, in order to reduce protocol overhead.

2. The detection delay (which is the longest period that can pass after one process terminates and before the heartbeat protocol causes all processes to terminate) should be small in order to increase protocol responsiveness.

3. The probability of premature termination (which is the probability that the heartbeat protocol causes all processes to terminate due to the loss of *beat* messages) should be small in order to increase protocol reliability.

These three objectives are somewhat contradictory. For example, to reduce both the rate of sending *beat* messages

*This work is supported in part by grant ARP-320 from the Texas Advanced Research Program.

and the detection delay in a heartbeat protocol, the protocol should cause all processes to terminate when a small number of *beat* messages are missed by their intended receivers. In this case, however, the probability of premature termination due to the loss of *beat* messages is relatively large. Therefore, every heartbeat protocol is a compromise between these contradictory objectives.

In this paper, we present a family of heartbeat protocols where a good compromise between these three objectives is achieved. We start our presentation by carefully defining the heartbeat problem.

2 The heartbeat problem

Consider a network of processes that exchange *beat* messages. At each instant, each process in the network is either active or inactive. An active process can later become inactive, but an inactive process remains inactive indefinitely. An active process can exchange *beat* messages with other processes in the network, whereas an inactive process can receive but not send *beat* messages. Each *beat* message that is sent to a process (whether active or inactive) is either eventually received by that process or lost.

It is required to design the processes in this network such that the following three conditions are satisfied.

1. Start: Initially, all processes in the network are active.
2. Choice: Periodically, each active process chooses whether to remain active or become inactive.
3. Outcome: If every process in the network continues to choose to remain active, then all processes remain active indefinitely. On the other hand, if one or more processes ever choose to become inactive, then all processes in the network eventually become inactive.

Let $p[0], p[1], ..., p[n]$ be the processes in the network. In order to satisfy the start condition, each process $p[i]$ has a local boolean variable, named *active*, whose initial value is **true**.

To satisfy the choice condition, each process $p[i]$ has an action, called the activity action, of the following form.

active $\rightarrow$ **if true** $\rightarrow$ **skip**
$\quad\quad\quad$ [] **true** $\rightarrow$ *active* $:=$ **false**
$\quad\quad\quad$ **fi**

This action is enabled for execution as long as variable *active* of process $p[i]$ has the value **true**. Any execution of this action either keeps the current value (**true**) of variable *active* unchanged, or assigns variable *active* the value **false**. (The Abstract Protocol notation of [7] is used in defining the processes in this paper. However, for the rest of the paper, we assume that the reader is not familiar with that notation.)

To satisfy the outcome condition, the processes need to exchange *beat* messages according to some heartbeat protocol. In the following sections, we present a family of four heartbeat protocols.

1. The first protocol is called the binary heartbeat protocol. This protocol involves two processes, $p[0]$ and $p[1]$. No process can join or leave this protocol.
2. The second protocol is called the static heartbeat protocol. This protocol involves $n + 1$ processes, $p[0]$ to $p[n]$. No process can join or leave this protocol.
3. The third protocol is called the expanding heartbeat protocol. This protocol starts with only one process, $p[0]$. Each of the other processes can join the protocol later. No process can leave this protocol.
4. The fourth protocol is called the dynamic heartbeat protocol. This protocol starts with only one process, $p[0]$. Each of the other processes can join the protocol later. Each process that joins the protocol can leave afterwards.

3 The binary heartbeat protocol

Consider the case where the network that has only two processes, $p[0]$ and $p[1]$. The communication between $p[0]$ and $p[1]$ can be partitioned into successive time periods. In each period, process $p[0]$ sends a *beat* message to process $p[1]$ then waits to receive back a *beat* message from $p[1]$. The length of each period depends on the events that occurred in the preceding period according to the following three rules.

1. If in a period, $p[0]$ sends a *beat* message to $p[1]$ and receives a *beat* message from $p[1]$, then $p[0]$ makes the length of the next period a large value *tmax* (irrespective on the length of the current period).
2. If in a period, $p[0]$ sends a *beat* message to $p[1]$ but does not receive a *beat* message from $p[1]$, then $p[0]$ makes the length of the next period half that of the current period.
3. If the length of the next period ever becomes less than a specified value *tmin*, that is an upper bound on the round-trip delay between $p[0]$ and $p[1]$, then $p[0]$ becomes inactive and stops sending *beat* messages to $p[1]$.

Some explanation concerning these three rules is in order. Rule 1 is adopted to ensure that when process $p[0]$ and $p[1]$ and the communication medium between them are all up (a typical situation), the rate of sending *beat* messages is kept small. Rules 2 and 3 are adopted so that when $p[0]$ suspects a failure or termination, $p[0]$ tries to refute this

suspicion several times in a short span before it finally accepts its suspicion and becomes inactive. Thus, these two rules ensure that both the detection delay and the probability of premature termination are kept small. In effect, the three rules constitute our compromise between the conflicting design objectives of heartbeat protocols discussed in Section 1.

From these three rules, if $p[0]$ does not receive any *beat* message for a period of 2*tmax*, then $p[0]$ becomes inactive. Moreover, if $p[1]$ does not receive any *beat* message for a period of 3*tmax* $-$ *tmin* (and so it does not send any *beat* messages for the same period), then $p[1]$ recognizes that $p[0]$ has already become inactive and $p[1]$ itself becomes inactive.

To explain the period 3*tmax* $-$ *tmin*, consider the following scenario.

1. $p[0]$ sends and receives *beat* messages. The period is *tmax*.
2. The network fails; all further messages are lost.
3. After a period of *tmax*, $p[0]$ sends a beat message.
4. After another period of *tmax*, $p[0]$ has received no *beat* message. It sends a *beat* message and makes the period *tmax*$/2$.
5. $p[0]$ continues to halve the period until it terminates.

The time between steps 1 and 3 is *tmax*, between steps 3 and 4 is *tmax*, and between steps 4 and 5 is bounded by *tmax* $-$ *tmin*. Thus, the period between steps 1 and 5 is bounded by 3*tmax* $-$ *tmin*.

In this binary heartbeat protocol, process $p[0]$ has two constants, *tmin* and *tmax*, and three variables named *active*, *rcvd*, and *t*. Variable *active* is discussed in Section 2. Variable *rcvd* is used to indicate whether $p[0]$ has received a *beat* message from $p[1]$ in the current period. Variable *t* stores the length of the current period. Process $p[0]$ can be defined as follows.

process $p[0]$
const *tmin*, *tmax* : **integer** $\{0 < tmin \leq tmax\}$
var *active* : **boolean**, $\{$initially **true**$\}$
rcvd : **boolean**, $\{$initially **true**$\}$
t : $0..tmax$ $\{$initially *tmax*$\}$
begin
active $\rightarrow$ **if true** $\rightarrow$ **skip**
$[]$ **true** $\rightarrow$ *active* $:=$ **false**
fi
$[]$ **timeout** *active* $\wedge$
$\{$a time period of at least *t* units has passed without sending a *beat* message$\}$ $\rightarrow$
if *rcvd* $\rightarrow$ $t := tmax$
$[]$ $\neg$*rcvd* $\rightarrow$ $t := t/2$
fi;
if $t < tmin \rightarrow$ *active* $:=$ **false**
$[]$ $t \geq tmin \rightarrow$ **send** *beat* **to** $p[1]$;
rcvd $:=$ **false**
fi
$[]$ **rcv** *beat* **from** $p[1] \rightarrow$ **if** *active* $\rightarrow$ *rcvd* $:=$ **true**
$[]$ $\neg$*active* $\rightarrow$ **skip**
fi
end

Process $p[0]$ has three actions. The first action is the activity action discussed in Section 2. The second action is a timeout action that is enabled for execution when $p[0]$ is active and the current period has ended. Executing this action consists of computing length *t* of the next period and deciding whether to start the next period (if $t >$ *tmin*), or become inactive (if $t \leq$ *tmin*). In the third action, $p[0]$ receives a *beat* message from $p[1]$ and assigns its variable *rcvd* the value **true**.

Process $p[1]$ in the binary heartbeat protocol can be defined as follows.

process $p[1]$
const *tmin*, *tmax* : **integer** $\{0 < tmin \leq tmax\}$
var *active* : **boolean** $\{$initially **true**$\}$
begin
active $\rightarrow$ **if true** $\rightarrow$ **skip**
$[]$ **true** $\rightarrow$ *active* $:=$ **false**
fi
$[]$ **rcv** *beat* **from** $p[0] \rightarrow$
if *active* $\rightarrow$ **send** *beat* **to** $p[0]$
$[]$ $\neg$*active* $\rightarrow$ **skip**
fi
$[]$ **timeout** *active* $\wedge$
$\{$a time period of at least 3*tmax* $-$ *tmin* units has passed without receiving a *beat* message$\}$ $\rightarrow$
active $:=$ **false**
end

Process $p[1]$ has three actions. The first action is the activity action discussed in Section 2. In the second action, $p[1]$ receives a *beat* message from $p[0]$ then sends a *beat* message to $p[0]$ (if $p[1]$ is still active). In the third action, $p[1]$ recognizes that it is still active and that a time period of 3*tmax* $-$ *tmin* has passed without receiving any *beat* message. In this case, $p[1]$ becomes inactive.

It is instructive to compare the binary heartbeat protocol with another heartbeat protocol, called the two-phase heartbeat protocol. In the two-phase heartbeat protocol, as long as process $p[0]$ keeps on receiving the expected *beat* messages from process $p[1]$, $p[0]$ sends a *beat* message to $p[1]$ every *tmax* time units. However, if process $p[0]$ does not receive an expected *beat* message from process $p[1]$,

then $p[0]$ starts to send a *beat* message to $p[1]$ every *tmin* time units.

To see that the binary heartbeat protocol is more efficient than the two-phase heartbeat protocol, consider the following three-step scenario:

1. $p[0]$ sends a *beat* message, then does not receive a *beat* message.

2. $p[0]$ sends a *beat* message, then receives a *beat* message.

3. $p[0]$ sends a *beat* message, then receives a *beat* message.

In the binary heartbeat protocol, the time period between steps 1 and 2 is *tmax*, and the time period between steps 2 and 3 is $tmax/2$. Thus, the binary heartbeat protocol executes these three steps in ($tmax + tmax/2$) time units. In the two-phase heartbeat protocol, the time period between steps 1 and 2 is *tmax* and the time period between steps 2 and 3 is *tmin*. Thus, the two-phase heartbeat protocol executes these three steps in ($tmax + tmin$) time units, which is usually less than ($tmax + tmax/2$). This shows that in the same time period, the binary heartbeat protocol sends a smaller number of messages than the two-phase heartbeat protocol whenever a *beat* message is lost.

The Transmission Control Protocol (or TCP for short) has an optional heartbeat feature based on the two-phase heartbeat protocol discussed above[10]. In this case, the values of *tmax* and *tmin* are chosen very large (namely, $tmax = 2$ hours, and $tmin = 75$ seconds) to compensate for the inherent inefficiency of this protocol.

It is also instructive to compare the binary heartbeat protocol with the timeout mechanism in TCP. In TCP, a process $p[0]$ sends a sequence of data messages to another process $p[1]$, then waits to receive an acknowledgment from $p[1]$ for each sent data message. If $p[0]$ does not receive an acknowledgment for some data message, then $p[0]$ resends the data message after some timeout period *t* and makes the timeout period $2t$. Thus, a data message can be resent from $p[0]$ to $p[1]$ after *tmin*, then after $2tmin$, then after $4tmin$, ..., and finally after *tmax* time units. Clearly, this exponential back-off in the TCP timeout mechanism is different from the exponential speed-up in the binary heartbeat protocol. The reason for this difference is two-fold.

1. TCP attempts to send data messages as fast as possible; hence, its timeout period is initially *tmin*. On the other hand, the binary heartbeat protocol attempts to send as few *beat* messages as possible; its time period between sending two successive *beat* messages is initially *tmax*.

2. In TCP, the timeout period between sending a data message and resending it is very short initially. Hence, this timeout period needs to be lengthened (by a factor of two) to prevent the same congestion that caused loss of the first message from causing loss of the second message. On the other hand, in the binary heartbeat protocol, the time period between sending two successive *beat* messages is very long initially. Hence, only with a small probability can the same congestion cause the loss of both messages, even if this time period is shortened (by a factor of two).

4 Analysis of the binary heartbeat protocol

If either process ($p[0]$ or $p[1]$) in the binary heartbeat protocol executes its first action and assigns its variable *active* the value **false**, then eventually the other process times out and assigns its variable *active* the value **false**, in accordance with the outcome requirement in Section 2. However, because sent *beat* messages can be lost, it is also possible that both processes time out and assign their variables *active* the value **false**, in violation of the outcome requirement. This undesirable possibility cannot be avoided as long as the probability of message loss is non-zero. Fortunately, as discussed in this section, the values of the two constants *tmin* and *tmax* in the binary heartbeat protocol can be chosen to ensure that the probability of this possibility is small. We start our discussion by introducing the concept of rounds.

A round is a sequence of all events that occur during an execution of the binary heartbeat protocol starting at the instant before $p[0]$ sends a *beat* message and ending at the instant before $p[0]$ sends the next *beat* message.

If in a round process $p[0]$ receives a *beat* message, then the round is called complete. Otherwise, the round is called incomplete. Let *R* denote the maximum number of consecutive incomplete rounds that can occur during any execution of the binary heartbeat protocol.

Consider *R* consecutive incomplete rounds that occur during some execution of the protocol. In these *R* rounds, the first round takes *tmax* time units, the second round takes $tmax/2$ time units, the third round takes $tmax/4$ time units, and so on. Thus, the *R*-th round takes $tmax/2^{R-1}$ time units, where $tmin \leq tmax/2^{R-1}$. Moreover, because the *R*-th round is the last one, we have $tmin > tmax/2^R$. Therefore, we end up with the following relation involving *tmin*, *tmax*, and *R*.

$$2^{R-1} * tmin \leq tmax < 2^R * tmin \qquad (1)$$

A complete round is called terminal iff it is followed by *R* incomplete rounds caused by the loss of sent *beat* messages. If a terminal round occurs during any execution of the binary heartbeat protocol, then the protocol terminates

after the terminal round and the *R* incomplete rounds that follow it. Such a termination is premature and should be avoided as much as possible. Hence, the probability that a round is terminal should be kept very small.

The probability P.terminal that a round is terminal can be computed as follows.

P.terminal = probability that a round is terminal
= probability that the next *R* rounds are incomplete due to the loss of sent *beat* messages
= (probability that *beat* messages are lost in a round)^R
= (1 - probability that no *beat* message is lost in a round)^R
= (1 - (1 - probability that one *beat* message is lost)^2)^R

Therefore, we get the following relation involving *R*, P.terminal, and P.loss, where P.loss is the probability that a sent (*beat*) message is lost. (For convenience, we assume that P.loss is constant and independent of past losses.)

$$\text{P.terminal} = (1 - (1 - \text{P.loss})^2)^R \tag{2}$$

Because the probability that a complete round is terminal is non-zero (though very small), any execution of the binary heartbeat protocol can terminate prematurely. Next, we compute the probability of premature termination.

Consider an execution of the binary heartbeat protocol where mature termination occurs within T time units. The number r of complete rounds that can occur during this execution is at most $T/$*tmax*. Any round i in this execution, where $1 \leq i \leq r-2$, can be terminal. (Note that a terminal round is followed by *R* incomplete rounds that take 2*tmax* time units. Therefore, neither round $r-1$ nor round r can be terminal.) If round i is terminal, then each previous round (namely rounds 1, 2, ..., and $i-1$) is non-terminal. Hence, the probability, that round i is terminal, is $((1 - \text{P.terminal})^{i-1} * \text{P.terminal})$. The occurrence of a terminal round signals premature termination. Therefore, the probability P.premature of premature termination for any execution of the binary heartbeat protocol, where mature termination occurs within T time units and $r = T/$*tmax*, is as follows.

$$\text{P.premature} = \begin{cases} 0 & \text{if } r \leq 2 \\ \sum : 1 \leq i \leq r-2 : & \\ \quad (1 - \text{P.terminal})^{i-1} * \text{P.terminal} & \text{if } r > 2 \end{cases} \tag{3}$$

The three relations (1), (2), and (3) can be used in the following procedure to compute *tmin*, *tmax*, and *R* and to ensure that P.terminal and P.premature are small.

1. Estimate the values of *tmin* and P.loss from the characteristics of the network where the binary heartbeat protocol is to be implemented. (Recall that *tmin* is an upper bound on the round-trip delay for a *beat* message, and P.loss is the probability that a sent *beat* message is lost.)

2. Choose an acceptable value for the detection delay D, which is the time period from the instant when termination occurs to the instant when it is detected. Then compute *tmax* as $D/3$.

3. Use the estimated *tmin* and the computed *tmax* to compute *R* from relation (1).

4. Use the estimated P.loss and the computed *R* to compute P.terminal from relation (2).

5. Compute P.premature for a given T as follows. First, use the computed *tmax* and the given T to compute r as $T/tmax$. Second, use the computed P.terminal and r to compute P.premature from relation (3).

Next, we apply this procedure in two situations. In the first situation, the binary heartbeat protocol is to be implemented in a local area network. In the second situation, the protocol is to be implemented in a wide area network.

In a local area network, we estimate that *tmin* $= 1$ second and P.loss $= .0001$. Choosing the detection delay D to be 60 seconds, we get *tmax* $= 20$ seconds. From relation (1), we compute $R = 5$. Therefore, P.terminal is about $1/(3 * 10^{18})$ from relation (2). For $T = 1$ hour, we get $r = 180$ and P.premature $= 1/(2 * 10^{16})$ from relation (3).

In a wide area network, we estimate that *tmin* $= 10$ seconds and P.loss $= .1$. Choosing the detection delay D to be 18 minutes, we get *tmax* $= 6$ minutes. From relation (1), we compute $R = 6$. Therefore, P.terminal is about $1/21000$ from relation (2). For $T = 1$ hour, we get $r = 10$ and P.premature $\approx 1/2700$ from relation (3).

5 The static heartbeat protocol

The binary heartbeat protocol can be extended to a protocol that involves $n+1$ processes, $p[0]$ to $p[n]$. The extended protocol is called the static heartbeat protocol.

In the static heartbeat protocol, process $p[0]$ executes a binary heartbeat protocol with every process $p[i]$, where $1 \leq i \leq n$. Thus, the communication between $p[0]$ and the $p[i]$ processes can be partitioned into periods. In each period, process $p[0]$ sends a *beat* message to every $p[i]$ process then waits to receive a *beat* message from every $p[i]$ process. When $p[0]$ receives a *beat* message from any $p[i]$, $p[0]$ records this fact by assigning its element *rcvd*$[i]$ the value **true**.

At the end of each period, process $p[0]$ computes the length of the next period as follows. First, $p[0]$ computes the length of the next period *tm*$[i]$ for each process $p[i]$, assuming a binary heartbeat protocol between $p[0]$ and $p[i]$.

Second, $p[0]$ selects the smallest *tm*$[i]$ to be the length *t* of the next period:

$$t = min(tm[1], tm[2], ..., tm[n])$$

Process $p[0]$ in the static heartbeat protocol can be defined as follows. (Note that the three actions in this $p[0]$ correspond to the the three actions in $p[0]$ of the binary heartbeat protocol.)

process $p[0]$
const *tmin*, *tmax* : **integer** $\{0 < tmin \leq tmax\}$
var *active* : **boolean**, {initially **true**}
rcvd : **array**$[1..n]$ **of boolean**, {initially **true**}
tm : **array**$[1..n]$ **of** $0..tmax$, {initially *tmax*}
t : $0..tmax$, {initially *tmax*}
$k : 1..n+1$
par $i : 1..n$
begin
active $\rightarrow$ **if true** $\rightarrow$ **skip**
[] **true** [] *active* := **false**
fi
[] **timeout** *active* $\wedge$
{a time period of at least *t* units has passed
without sending a *beat* message} $\rightarrow$
$k := 1$;
do $k \leq n \rightarrow$ **if** *rcvd*$[k] \rightarrow$ *tm*$[k]$:= *tmax*
[] $\neg$*rcvd*$[k] \rightarrow$ *tm*$[k]$:= *tm*$[k]/2$
fi; $k := k+1$
od; $t :=$ **MIN**(*tm*);
if $t <$ *tmin* $\rightarrow$ *active* := **false**
[] $t \geq$ *tmin* $\rightarrow$ **BCAST**
fi
[] **rcv** *beat* **from** $p[i] \rightarrow$ **if** *active* $\rightarrow$ *rcvd*$[i]$:= **true**
[] $\neg$*active* $\rightarrow$ **skip**
fi
end

In the second action, function **MIN**(*tm*) computes the value of the smallest element in array *tm*. Also, statement **BCAST** is defined as follows.

BCAST :: $k := 1$;
do $k \leq n \rightarrow$ **send** *beat* **to** $p[k]$;
rcvd$[k], k :=$ **false**, $k+1$
od

Each of the processes $p[0], ..., p[n]$ in the static heartbeat protocol is defined exactly as process $p[1]$ in the binary heartbeat protocol.

The analysis of the static heartbeat protocol is similar to that of the binary heartbeat protocol in Section 4. In particular, relations (1) and (3) of the binary heartbeat protocol are still valid for the static heartbeat protocol. Moreover, relation (2) of the binary heartbeat protocol can be modified for the static heartbeat protocol to become as follows.

$$\text{P.terminal} = n * (1 - (1 - \text{P.loss})^2)^R \qquad (4)$$

6 The expanding heartbeat protocol

In order to demonstrate that this protocol can be used in a flexible environment, we extend the static heartbeat protocol as follows. Initially, only process $p[0]$ is involved in the heartbeat protocol. Later, any process $p[i]$, where $1 \leq i \leq n$, can join the protocol by sending a *beat* message to $p[0]$ every *tmin* time units. This continues until $p[i]$ starts receiving *beat* messages from $p[0]$ and recognizes that it has joined the heartbeat protocol. When this happens, $p[i]$ stops sending any *beat* message to $p[0]$ unless as a reply to a received *beat* message from $p[0]$. This protocol is called the expanded heartbeat protocol.

Process $p[0]$ in the expanded heartbeat protocol has one additional array, named *jnd*, over process $p[0]$ in the static heartbeat protocol. Array *jnd* is declared as follows.

var *jnd* : **array**$[1..n]$ **of boolean** {initially **false**}

For every i, where $1 \leq i \leq n$, the value of *jnd*$[i]$ is true iff $p[0]$ has ever received a *beat* message from $p[i]$ indicating that $p[i]$ is trying to join or has joined the heartbeat protocol. Process $p[0]$ in the expanding heartbeat protocol can be defined as follows.

process $p[0]$
const *tmin*, *tmax* : **integer** $\{0 < tmin \leq tmax\}$
var *active* : **boolean**, {initially **true**}
rcvd : **array**$[1..n]$ **of boolean**, {initially **true**}
jnd : **array**$[1..n]$ **of boolean**, {initially **false**}
tm : **array**$[1..n]$ **of** $0..tmax$, {initially *tmax*}
t : $0..tmax$, {initially *tmax*}
$k : 1..n+1$
par $i : 1..n$
begin
active $\rightarrow$ **if true** $\rightarrow$ **skip**
[] **true** $\rightarrow$ *active* := **false**
fi
[] **timeout** *active* $\wedge$
{a time period of at least t units has passed
without sending a *beat* message} $\rightarrow$
$k := 1$;
do $k \leq n \rightarrow$
if $\neg$*jnd*$[k] \rightarrow$ **skip**
[] *jnd*$[k] \wedge$ *rcvd*$[k] \rightarrow$ *tm*$[k]$:= *tmax*
[] *jnd*$[k] \wedge \neg$*rcvd*$[k] \rightarrow$ *tm*$[k]$:= *tm*$[k]/2$
fi; $k := k+1$
od; $t :=$ **MIN**(*tm*);
if $t <$ *tmin* $\rightarrow$ *active* := **false**
[] $t \geq$ *tmin* $\rightarrow$ **BCAST**$'$

```
    fi
 [] rcv beat from p[i] → if active → rcvd[i] := true;
                                     jnd[i] := true
                          [] ¬active → skip
                         fi
end
```

In the second action, statement **BCAST′** is defined as follows.

```
BCAST′ :: k := 1;
          do k ≤ n → if jnd[k] → send beat to p[k]
                     [] ¬jnd[k] → skip
                     fi; rcvd[k], k := false, k + 1
          od
```

Processes $p[1], ..., p[n]$ in the expanding heartbeat protocol have one additional variable, named *join*, over process $p[i : 1..n]$ in the static heartbeat protocol. Variable *join* in process $p[i]$ has the value **true** iff $p[i]$ has already joined the heartbeat protocol. Process $p[i : 1..n]$ in the expanding heartbeat protocol can be defined as follows.

```
process p[i : 1..n]
const tmin, tmax : integer                    {0 < tmin ≤ tmax}
var active : boolean,                          {initially true}
    join : boolean                             {initially false}
begin
  active → if true → skip
           [] true → active := false
           fi
 [] timeout active ∧ ¬join ∧
    {a time period of at least tmin units has passed
    without sending a beat message} →
      send beat to p[0]
 [] rcv beat from p[0] → if active → send beat to p[0];
                                     join := true
                         [] ¬active → skip
                         fi
 [] timeout active ∧
    {a time period of at least 3tmax − tmin units has
    passed without receiving a beat message} →
      active := false
end
```

7 The dynamic heartbeat protocol

In this section, we discuss how to extend the expanding heartbeat protocol to allow each process $p[i]$, where $1 \leq i \leq n$, to leave the heartbeat protocol anytime after it has joined it. The extended protocol is called the dynamic heartbeat protocol.

In the dynamic heartbeat protocol, each *beat* message has a boolean field. For a process $p[i]$ to join the heartbeat protocol, $p[i]$ sends a *beat*(**true**) message to $p[0]$ every *tmin* time units. When $p[i]$ joins the heartbeat protocol, the communication between $p[0]$ and $p[i]$ proceeds in periods. In each period, $p[0]$ sends a *beat*(**true**) message to $p[i]$ which replies by sending back a *beat*(**true**) message to $p[0]$. This continues until $p[i]$ decides to leave the heartbeat protocol. When this happens, $p[i]$ replies to each *beat*(**true**) message from $p[0]$ by sending back a *beat*(**false**) message to $p[0]$. When $p[0]$ receives a *beat*(**false**) message from process $p[i]$, $p[0]$ records the fact that $p[i]$ is no longer part of the heartbeat protocol (by assigning *jnd*$[i]$ the value **false**) and refrains from sending any more *beat*(**true**) messages to $p[i]$. When $p[i]$ detects that a time period of $3tmax - tmin$ has passed without receiving any *beat* messages, $p[i]$ becomes inactive (without causing any other process to become inactive).

Process $p[0]$ in the dynamic heartbeat protocol is similar to process $p[0]$ in the expanding heartbeat protocol except for two modifications. First, $p[0]$ in the dynamic heartbeat protocol sends a *beat*(**true**) message, instead of a *beat* message, to every $p[i]$ where *jnd*$[i]$ = **true**. Second, when $p[0]$ in the dynamic heartbeat protocol receives a *beat*(b) message from $p[i]$, $p[0]$ assigns both *rcvd*$[i]$ and *jnd*$[i]$ the value b.

Processes $p[1], ..., p[n]$ in the dynamic heartbeat protocol can be defined as follows.

```
process p[i : 1..n]
const tmin, tmax : integer                    {0 < tmin ≤ tmax}
var active : boolean                           {initially true}
    join : boolean,                            {initially false}
    leave : boolean                            {initially false}
begin
  active → if true → skip
           [] true → active := false
           fi
 [] timeout active ∧ ¬join ∧
    {a time period of at least tmin units has passed
    without sending a beat message} →
      send beat(true) to p[0]
 [] rcv beat(true) from p[0] →
      if active ∧ ¬leave → send beat(true) to p[0];
                                     join := true
       [] active ∧ leave → send beat(false) to p[0]
       [] ¬active → skip
      fi
 [] active ∧ join ∧ ¬leave → leave := true
 [] timeout active ∧
    {a time period of at least 3tmax − tmin units has
    passed without receiving a beat message} →
      active := false
end
```

8 Concluding remarks

In this paper, we have presented a family of heartbeat protocols that achieve a good compromise between three contradictory objectives: a small sending rate of *beat* messages, a small detection delay, and a small probability for premature termination. For each protocol in this family, the rate of sending *beat* messages is $1/tmax$, the detection delay is $3tmax - tmin$, and the probability of premature termination is as defined by relation (2) in Section 4 or relation (4) in Section 5.

In our protocols, we assumed that the processes are arranged in a flat spanning tree where $p[0]$ is the parent and each of the other processes $p[1], ..., p[n]$ is a child. Clearly these protocols can be extended in a straightforward manner to the case where the processes are arranged in a general spanning tree. In the extended protocols, each process acts as $p[0]$ when it is communicating with its children in the tree and acts as a process $p[i : 1..n]$ when it is communicating with its parent in the tree. This change will limit the n factor in relation (4) at the cost of increasing the detection delay.

Currently, we are implementing these heartbeat protocols on the Internet. Our implementation is based on a C library that we have developed to support fast prototyping of network protocols from their abstract specifications[8].

Acknowledgments

We are thankful to Anish Arora and the anonymous referees for their helpful comments on an earlier version of this paper.

References

[1] Barborak, M., M. Malek, and A. Dahbura, "The Consensus Problem in Fault-Tolerant Computing", *ACM Computing Surveys*, Vol. 25, No. 2, pp. 171-220, June 1993.

[2] Braden, R., editor, "Requirements for Internet Hosts-Communication Layers", *RFC 1122*, October 1989.

[3] Chandy, K. M. and J. Misra, "Termination Detection", Chapter 9 in *Parallel Program Design: A Foundation*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[4] Cristian, F., "Reaching Agreement on Processor-Group Membership", *Distributed Computing*, Vol. 4, pp. 175-187, 1991.

[5] Dijkstra, E. W. and C. S. Scholten, "Termination Detection for Diffusing Computations", *Information Processing Letters*, Vol. 11, No. 1, August 1980.

[6] Francez, N., "Distributed Termination", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 1, January 1980.

[7] Gouda, M. G. "Network Processes", Chapter 3 in *Elements of Network Protocol Design*, John Wiley and Sons, 1998.

[8] McGuire, T., "Implementing Abstract Protocols in C". M. A. Thesis, Department of Computer Sciences, the University of Texas at Austin, 1994. Also appeared as a Technical Report, TR 96-31, Department of Computer Sciences, the University of Texas at Austin, 1996.

[9] Pradhan, D. K. et. al., "Recoverable Mobile Environment Design and Trade-off Analysis", *FTCS '96*, June 1996.

[10] Stevens, R. W., *TCP/IP Illustrated: The Protocols*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[11] Tseng, Y. C., "Detecting Termination By Weight Throwing in a Faulty Distributed System", *Journal of Parallel and Distributed Computing*, Vol. 25, No. 1, February 1995.

[12] Vogels, W., "World Wide Failures", *ACM SIGOPS 1996 European Workshop*, September 1996.