

# Chapter 1

## Termination Detection Service

### 1.1 Introduction

Given a generic distributed computation, say  $X$ , the goal in termination detection is to obtain another (distributed) computation, say  $Y$ , that executes alongside  $X$  on the same platform and signals if and when  $X$  has terminated. More precisely, each process in  $X$  is either “active” or “inactive”. An active process can do local computation, send and receive messages, and become inactive. An inactive process does nothing except become active upon receiving a message. The computation  $X$  is said to have terminated if all its processes are inactive and none of its messages are in transit. It is not trivial for  $Y$  to detect termination of  $X$  because  $Y$  executes on the same platform as  $X$ , and hence suffers from the same atomicity limitations. Naturally,  $Y$  must not affect  $X$ .

This chapter formalizes the termination detection problem as a service, just like the lock service formalizes the mutual exclusion problem. Specifically, the termination detection service is an enhanced message-passing service in which, in addition to message transmit and receive events, each user can signal the offer when it becomes inactive, and the offerer signals a particular user if and when all users have signalled inactivity and no messages are in transit. Figure 1.1 illustrates the relationship of the termination detection service with respect to regular message-passing services.

The specification here provides fifo channels, but other kinds of channels can be just as easily specified. Later chapters will show distributed systems that offer the termination detection service with fifo channels making use of regular fifo channels.

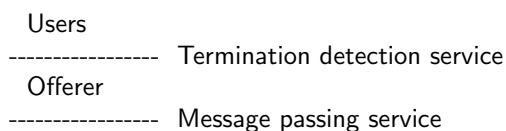


Figure 1.1: Termination detection layer above regular message-passing service.

### 1.2 Service specification

Below,  $Ids$  denotes the user ids,  $Msg$  denotes the messages that can be exchanged between the users, and  $active\_init$  is the initial active status of the users. To capture fifo message passing, we have the usual history variables:  $txh[u, v]$ , the sequence of messages transmitted by  $u$  to  $v$ , and  $rxh[u, v]$ , the sequence of messages received by  $u$  from  $v$ .

```
service-program TermnDetctnSrv( Ids, Msg, boolean[Ids] active_init ) {  
  init() {  
    boolean[Ids] active := active_init;
```

```

MsgSeq[Ids, Ids] txh, rxh := ⟨⟩; // transmit and receive histories
boolean tsig := false; // true iff termination has been signalled
}

function boolean tstate() { // termination state
  [forall Ids u,v : ¬active[u] and txh[u,v]=rxh[v,u]]
}

dnw Tx( Ids u, v, Msg msg ) {
  ec active[u] and (u≠v)
  ac txh[u,v] := txh[u,v]◦⟨msg⟩;
}

upw Rx( Ids u, v, Msg msg ) {
  ec (u≠v) and ((rxh[u,v]◦⟨msg⟩) prefix-of txh[v,u])
  ac rxh[u,v] := rxh[u,v]◦⟨msg⟩;
  active[u] := true;
}

dnw Inactive( Ids u ) {
  ec active[u]
  ac active[u] := false;
}

upw TermnDetctd() {
  ec tstate() and ¬tsig
  ac tsig := true;
}

progress-obligation() {
  [ forall Ids u,v : txh[u,v].size≥i ⇔ rxh[v,u].size≥i ]; // every message sent is eventually received

  tstate() ⇔ tsig; // termination is eventually signalled
}
} // termination detection service

```

Note that the upward TermnDetctd() event can be mapped to any one user.

### 1.3 Concluding Remarks

TBD

### 1.4 Exercises

- 1.1 Modify the termination service in two ways. First, it is started by a specific user, say user  $u_0$ . Second, once started, it eventually signals whether or not termination has been detected. (If it signals that termination has not been detected, the user would presumably issue a new start at some later point.)

## Chapter 2

# Termination Detection for Diffusing Computations

### 2.1 Introduction

A **diffusing** computation is a distributed computation that starts with exactly one active process. This simple restriction permits a very efficient and elegant distributed solution of termination detection [?]. The solution implements a dynamic tree that includes all active processes and is rooted at the process that is initially active. The tree grows whenever a node not on the tree becomes active and shrinks whenever a leaf node becomes inactive. A distributed dynamic tree, and hence the algorithm underlying the solution, is useful for many purposes other than termination detection, for example, traversing the nodes such that each node is visited exactly once.

This chapter describes the solution and casts it into a system that offers the termination detection service for diffusing computations.

### 2.2 Solution

The termination detection algorithm maintains a dynamic tree rooted at the process that is initially active. Let this process be identified as  $u_0$ . A process not on this tree is inactive and has no outgoing messages in transit. So when process  $u_0$  finds the tree to be empty, it can signal termination detection. The tree grows by an edge  $\langle u, v \rangle$  when (and only when) a node  $u$  in the tree activates a node  $v$  not in the tree. The tree shrinks when a leaf node  $v$  in the tree is inactive and has no outgoing messages in transit. The edge  $\langle u, v \rangle$  that is removed is the one that was introduced when  $v$  joined the tree. Allowing only a leaf node to leave ensures that the tree remains connected.

Initially only node  $u_0$  is in the tree. When a node  $v$  that is not on the tree receives a message from node  $u$  say, edge  $\langle u, v \rangle$  is added to the tree and node  $v$  becomes an active leaf node of the tree. Node  $v$  can evolve in three possible ways. The first possibility is that  $v$  stays active forever, in which case it stays in the tree forever (and so does  $u$ ). The second possibility is that  $v$  becomes inactive without having sent any messages, in which case throughout this active period  $v$  remains a leaf node and has no outgoing messages in transit. So  $v$  can leave the tree when it becomes inactive. The third possibility is that  $v$  sends messages and then becomes inactive, at which point  $v$  may have outgoing messages still in transit or it may have become a non-leaf node of the tree. So  $v$  can leave the tree only after it determines that these conditions do not hold.

So we want a mechanism by which any node  $v$  in the tree can detect (1) that it has no outgoing messages in transit, and (2) that it has no successor in the tree. Regarding the first condition, it suffices if every node sends an ack in response to each message reception. Then  $v$  would detect that no outgoing messages are in transit whenever it has received an ack for every message it has sent. Regarding the second condition, when a message  $m$  of  $v$  causes a node  $w$  to join the tree,  $w$  can indicate this in the ack it sends, i.e., a “tree-joining” ack. When  $w$  leaves the tree at some later point, it can send a “tree-leaving” intimation to  $v$ . Node  $v$  can leave only after receiving this intimation.

The above works but it can be simplified further. When  $w$  becomes a successor to  $v$ , instead of sending a tree-joining ack immediately and an tree-leaving intimation when  $w$  leaves the tree,  $w$  can simply withhold sending the ack until it leaves the tree. In this case, the tree-leaving intimation is not needed and the tree-joining ack can simply be a regular ack. Also, note that an ack need not identify the message acked, so sequence numbers are not needed.

We now summarize the algorithm. Node  $u$  maintains the following variables: **engaged**, a boolean indicating whether  $u$  is in the tree; **engager**, identifying the node that last activated  $u$  (if **engaged** is true); and **tod**, the total number of unacked messages that  $u$  has sent. Initially, **engaged** is true for  $u_0$  and false for every other node, and **tod** is zero for all nodes.

Each node  $u$  executes the following rules:

```
Upon reception of a message from node v
  if not engaged
    then { engaged := true ; engager := v }
    else send ack to v;
```

```
Upon transmission of a message
  tod := tod + 1;
```

```
Upon reception of an ack
  tod := tod - 1;
```

```
When (inactive and tod=0 and engaged) do
  if  $u \neq u_0$  then {
    send ack to engager;
    engaged := false;
  }
  else signal termination detected
```

The analysis is straightforward given the above discussion. We first define some terms. A node is *engaged* if its **engaged** is true. A directed edge  $\langle u, v \rangle$  is an *engagement edge* if  $v$  is engaged and  $v$ 's **engager** is  $u$ . Let  $E$  be the directed graph whose vertices are the engaged nodes and whose arcs are the engagement edges. The following predicates are invariant:

$A_1$  :  $E$  forms an out-tree rooted at node  $u_0$  (that is, every engaged node is reachable from node  $u_0$  via exactly one directed path).

$A_2$  : For any node  $u$ : **tod** = (number of outgoing messages of  $u$  in transit) + (number of incoming acks to  $u$  in transit) + (number of outgoing engagement edges from  $u$ ).

$A_3$  : For any node  $u$ : if  $u$  is not engaged then it is inactive and its **tod** equals zero.

$\square A_{1..3}$  holds because  $A_{1..3}$  holds initially and is preserved by each rule execution.  $A_{1..3}$  implies that if node  $u_0$  is inactive and its **tod** is zero then  $E$  consists only of  $u_0$  and the system has terminated. This establishes safety.

For progress, we need to show that termination leads to node  $u_0$ 's **tod** becoming zero. A suitable metric is the tuple  $\langle F, G \rangle$  under lexicographic ordering, where  $F$  is the number of nodes in  $E$  and  $G$  is the aggregate **tod** of the leaf nodes of  $E$ . The details are as follows. Assume termination. Assume  $u_0$ 's **tod** is not zero. Let  $u$  be a leaf node of  $E$  (at least one exists by  $\square A_1$ ). Node  $u$  has no outgoing messages in transit (because of termination) and no engagement edges (because it is a leaf node). So  $u$ 's **tod** equals the number of incoming acks in transit (from  $\square A_2$ ). Eventually all these acks are received (from channel progress) and so  $u$ 's **tod** becomes zero, after which  $u$  eventually leaves  $E$  (since it is inactive and remains so) unless  $u$  is  $u_0$ . Thus  $E$  keeps shrinking until it eventually consists of only  $u_0$  and its **tod** equals zero.

## 2.3 Offerer system

We now cast the above algorithm into a distributed system TDD that offers the termination detection service for diffusing computations using fifo channels. The component of the distributed system at location  $u$  is as follows, where  $\text{Msgx}$  is  $\text{Msg} \cup \{\text{ack}\}$ ,  $u_0$  is the initially active node id, and  $\text{outq}$  and  $\text{inq}$  are outgoing and incoming message queues.

```
system-program TDDcomp(Ids u, Ids u0, Msgx) { // TDD component at u
  // accesses TermnDetctnSrv(Ids, Msg, active_init) above and fifo(Ids, Msgx) below
```

```
  init() {
    MsgxId = RecordOf( Ids v; Msgx msg);
    MsgxIdSeq outq :=  $\langle \rangle$ ; // outgoing queue of messages awaiting transmission
    MsgxIdSeq inq :=  $\langle \rangle$ ; // incoming queue of messages awaiting delivery to local user
    boolean active; // true iff u is active
    boolean engaged; // true iff u is on the tree
    Ids engager; // node that last activated u (if engaged is true)
    Int tod := 0; // number of unacked messages that u has sent
    if u=u0 then {
      active := true;
      engaged := true;
    } else {
      active := false;
      engaged := false;
    }
  }
```

```
  xc-event Inactive()
    maps TermnDetctnSrv(Ids).Inactive(u)
    ec u in Ids
    ac active := false;
```

```
  xc-event Tx(Ids v, Msg msg)
    maps TermnDetctnSrv(Ids).Tx(u,v,msg)
    ec active
    ac append(outq, MsgxId(v, msg));
    tod := tod+1;
```

```
  xc-event Rx(Ids v, Msgx msg)
    maps fifo(Ids, Msgx).Rx(u,v,msg)
    ec true
    ac append(inq, MsgxId(v, msg));
```

```
  lc-event Deliver() { // to local user
    ec inq not empty
    ac Msgx x := remove(inq) ;
    if x.msg = ack
    then tod := tod - 1
    else {
      TermnDetctn(Ids,Msgs).Rx(u,v,x.msg);
      outq.append(v, Msgx(ack));
    }
  }
```

```
  lc-event ToChannel() {
    ec outq not empty
    ac Msgx x := remove(outq);
```

```

    fifo(Ids,Msgx).Tx(u,v,x.msg);

lc-event Disengage() {
  ec (not active) and (tod = 0) and engaged
  ac if (u = u0) then
    TermnDetctn(Ids).TermnDetctd()
  } else {
    engaged := false;
    outq.append(engager, Msgx(ack));
  }

fairness-assumptions { WFair of lc events }

} // end TDD component at u

```

Let  $TDD(Ids, Msg, Ids\ u_0)$  be the composite system of  $\{TDDcomp(u, u_0, Msg) : Ids\ u\}$ . Let  $active\_int$  be the boolean array on  $Ids$  with the entry for  $u_0$  being true and every other entry being false. Then  $TDD(Ids, Msg, u_0)$  offers  $TermnDetctnSrv(Ids, Msg, active\_int)$  using  $fifo(Ids, Msg)$ . The proof is left as an exercise.

## 2.4 Concluding Remarks

The algorithm described here is taken from [?] with some minor differences. What we call acks here are referred to as “signals” there. Our initially active node  $u_0$  is referred as the environment node there, and no process sends messages to the environment node during the computation.

The analysis is also taken from [?], with some significant differences. The analysis there develops the algorithm and desired invariants in incremental steps. The first desired invariant is that each edge has a non-negative deficit, where the deficit of an edge from  $u$  to  $v$  refers to the number of messages sent thus far from  $u$  to  $v$  minus the number of signals received thus far from  $v$  at  $u$ . This invariant is enforced by having each node maintain the deficit of each of its incoming edges and sending a signal back on an edge only if the edge’s deficit exceeds zero.

The next desired invariant is that for every node  $u$ , its total incoming deficit equals zero only if its total outgoing deficit equals zero. This is enforced by allowing a process to send a signal on an edge only if the incoming deficit on that edge is positive and either the total incoming deficit exceeds one or the total outgoing deficit equals zero. It is shown that the algorithm at this point ensures that if termination has occurred then eventually the environment node’s  $tod$  becomes zero.

Finally, the constraint is added that the final signal sent by a node is to its engager, thereby resulting in the final algorithm. (Another difference is that message transmission is assumed to be atomic with the corresponding reception, but this is not important because extending the analysis there to non-zero capacity channels is straightforward.)

## Chapter 3

# Distributed Snapshot Service

### 3.1 Introduction

Given a generic distributed computation, say  $X$ , the goal in snapshot computation is to obtain another (distributed) computation, say  $Y$ , that (1) can be started at any point, (2) executes alongside  $X$  on the same platform without disturbing  $X$ , and (3) constructs a global snapshot  $S$  corresponding to a global state of  $X$  at some point during  $Y$ 's computation.  $S$  consists of a snapshot of each process and channel of  $X$ .

This chapter formalizes the distributed snapshot problem as a service, similar to how the termination detection service formalized the termination detection problem. The distributed snapshot service is an enhanced message-passing service. In addition to the usual fifo message transmit and receive events, a particular user  $u_0$  can initiate the snapshot computation, following which the offerer eventually tells every other user when to take a snapshot of itself and provides each user with a snapshot of each incoming channel. The service allows  $u_0$  to initiate the snapshot computation as long as no snapshot computation is currently underway. Figure 1.1 illustrates the relationship of the distributed snapshot service with respect to regular message-passing services.

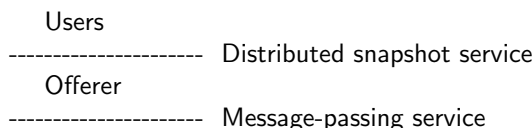


Figure 3.1: Distributed snapshot layer above regular message-passing service.

### 3.2 Service specification

Below,  $Ids$  denotes the user ids and  $Msg$  denotes the messages that can be exchanged between the users. The message-passing part of the service is modeled as usual, by transmit and receive events and history variables  $txh[u, v]$  (sequence of messages transmitted by  $u$  to  $v$ ) and  $rxh[u, v]$  (sequence of messages received by  $u$  from  $v$ ).

The snapshot part of the service involves three events:  $dwn\ Gss(u_0)$ , for user  $u_0$  to initiate a global snapshot computation;  $upw\ Uss(u)$ , for the offerer to tell user  $u$  (other than  $u_0$ ) to take its local snapshot; and  $upw\ Css(u, v, mseq)$ , for the offerer to provide user  $u$  with the snapshot  $mseq$  of messages in transit in the channel from  $v$  to  $u$ . The snapshot part of the service involves a history variable  $H$  that records the sequence of service event executions since initialization.  $H$  has entries of the form  $(GSS, u)$ ,  $(USS, u)$ ,  $(CSS, u, v, mseq)$ ,  $(TX, u, v, msg)$ ,  $(RX, u, v, msg)$ , where the first component indicates the type of event, the second component indicates the user that participates in the event, and the other components are self-explanatory. (Note that  $txh[u, v]$  and  $rxh[u, v]$  could be obtained as functions of  $H$ .)

Next, we define various functions for any sequence  $A$  of event execution signatures.

- Let  $\text{txg}(A, u, v)$  be the sequence of  $\text{Tx}(u, v, m)$  entries in  $A$ . Let  $\text{rxg}(A, u, v)$  be the sequence of  $\text{Rx}(u, v, m)$  entries in  $A$ . We say  $A$  is channel consistent if  $\text{rxg}(p, u, v)$  is a prefix of  $\text{txg}(p, u, v)$  for every prefix  $p$  of  $A$  and every pair of distinct ids  $u$  and  $v$ .
- Define  $A_{<}$  as follows:  $A_{<}$  equals  $A$  if  $A$  has no GSS entry; otherwise,  $A_{<}$  is the prefix of  $A$  upto (and excluding) the last GSS entry in  $A$ .
- Define  $A_{>}$  to be the complement of  $A_{<}$ , i.e.,  $A$  equals  $A_{<} \circ A_{>}$ .
- We say  $A$  ends in a complete snapshot, denoted  $\text{ecss}(A)$ , if  $A_{>}$  has exactly one  $\text{USS}(u)$  entry for every  $u$  other than  $u_0$  and exactly one  $\text{CSS}(u, v)$  entry for every pair of distinct ids  $u$  and  $v$ .
- We say  $A$  ends in an instantaneous snapshot, denoted  $\text{eiss}(A)$ , if
  - $A$  is channel-consistent.
  - $A$  ends in a complete snapshot.
  - For every pair of distinct ids  $u$  and  $v$ ,  $A_{>}$  has a  $\text{CSS}(u, v, \text{mseq})$  entry such that  $\text{rxg}(A_{<}, u, v) \circ \text{mseq} = \text{txg}(A_{<}, u, v)$ .
  - The GSS and USS entries in  $A_{>}$  are contiguous.
- We say  $A$  ends in a global snapshot, denoted  $\text{egss}(A)$ , if
  - $A$  is not empty.
  - $A_{>}$  can be reordered into  $\hat{A}_{>}$   $A_{>}$  can be reordered into  $\hat{A}_{>}$  such that  $B = A_{<} \circ \hat{A}_{>}$  ends in an instantaneous snapshot,
  - For every id  $u$ ,  $B$  and  $A$  have the same sequence of entries corresponding to events at  $u$ .

Given the above functions, the service specification is as follows. Essentially, if the realized history  $H$  ends in a snapshot,  $H_{>}$  should be reorderable to  $\hat{H}_{>}$  such that  $H_{<} \circ \hat{H}_{>}$  ends in an instantaneous snapshot.

```

service-program SnapshotSrv( Ids, Msg, Ids  $u_0$  ) {
  init() {
    int TX := 1, RX := 2, GSS := 3, USS := 4, CSS := 5; // event types
    ESig = RecordOf(
      int etype; // TX, RX, GSS, USS, CSS
      Ids xu; // location of event
      Ids xv if etype is TX, RX, CSS;
      MsgSeq mseq if etype is CSS;
    );

    ESigSeq H :=  $\langle \rangle$ ; // snapshot computation history
    MsgSeq[Ids, Ids] txh, rxh :=  $\langle \rangle$ ; // transmit and receive histories
  }

  dnw Tx( Ids u, v, Msg msg ) {
    ec  $u \neq v$ 
    ac txh[u,v] := txh[u,v]  $\circ \langle \text{msg} \rangle$ ;
    append(H, ESig(TX,u,v,msg));
  }

  upw Rx( Ids u, v, Msg msg ) {
    ec  $u \neq v$  and (( rxh[u,v]  $\circ \langle \text{msg} \rangle$  ) prefix-of txh[v,u])
    ac rxh[u,v] := rxh[u,v]  $\circ \langle \text{msg} \rangle$ ;
    append(H, ESig(RX,u,v,msg));
  }
}

```

```

dnw Gss( lds u0 ) {
  ec H has no GSS entry or ends in a complete snapshot
  ac append(H, ESig(GSS, u0));
}

upw Uss( lds u ) {
  ec u ≠ u0 and (H> has (GSS, u0) but not (USS, u))
  ac append(H, ESig(USS, u));
}

upw Css( lds u, lds v, MsgSeq mseq ) {
  ec (u ≠ v) and (H> has (GSS, u0) but not (CSS, u, v, *))
  and (ecss(H◦(CSS, u, v, mseq)) ⇒ egss(H◦(CSS, u, v, mseq)))
  ac append(H, ESig(CSS, u, v, mseq));
}

progress-obligation() {
  // every message sent is eventually received
  [ forall lds u, v : txh[u, v].size ≥ i ⇔ rxh[v, u].size ≥ i ];

  // every snapshot is eventually completed
  (H> not empty) ⇔ ecss(H)
}

} // snapshot service

```

### 3.3 Concluding Remarks

This snapshot service is more complicated than it needs to be. One way to make it simpler is to allow at most one snapshot to be taken (similar to the termination-detection service). But then it would not be helpful to solve your problem, other than using only its fifo channels service. Another way to make it simpler is to allow multiple snapshots to be taken concurrently, and index each one with a snapshot sequence number. This makes snapshot computations independent of each other. It would allow any node to initiate a global snapshot at any time. It would be simpler than the above because we would not need the special node  $u_0$  or the prefixing and suffixing operations.

**In summary: if you cannot understand the above snapshot service, then solve the problem using only the fifo channels of the service. If you can understand the above snapshot service, it can simplify your algorithm.**