

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.

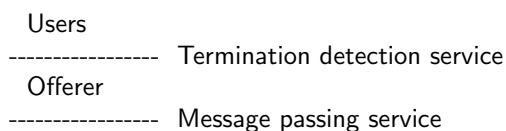


Figure 1.1: Termination detection layer above regular message-passing service.

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.)