Alert Communication Primitives in TCP

Mohamed G. Gouda
gouda@cs.utexas.edu
Department of Computer Science, The University of Texas at Austin
Taylor Hall 2.124, Austin, TX 78712

Tommy M. McGuire
mcguire@cs.utexas.edu
Department of Computer Science, The University of Texas at Austin
Taylor Hall 2.124, Austin, TX 78712

April 6, 2000

Abstract

We consider communication primitives that can be executed by an application process to exchange messages with another application process over a TCP/IP network. A communication primitive is called alert iff it satisfies two conditions. First, if during any execution of the primitive no failure occurs, then the execution completes successfully. Second, if during any execution of the primitive some failure occurs, then the execution is aborted and the process that initiated the execution is informed of the failure. Clearly alert communication primitives are useful in designing reliable distributed applications. We argue that the send primitive over TCP is alert, but the receive primitive over TCP is not. Then, we propose three new receive primitives over TCP and show that each of them is alert. We also discuss how to implement these three primitives and compare their performance.

Keywords: distributed applications, fault tolerance, Internet, interprocess communication, reliability, Transmission Control Protocol

1 Introduction

Consider a distributed application consisting of several processes that reside in different computers and communicate by exchanging messages over a TCP/IP network. In order to make each process in this application capable of detecting the failures of other processes, and the failures of the communication media between processes, the communication primitives that each process executes to communicate with other processes need to be alert in the following sense. Whenever a process *p* executes a communication primitive to exchange a message with another process *q*, then one of two outcomes is possible. If neither process *q* nor the communication medium between *p* and *q* fails (before execution of the primitive completes), then execution of the primitive completes successfully. Otherwise, execution of the primitive is aborted and process *p* is informed of some failure.

Clearly, the capability of each process *p* to detect the failure of any process *q* (with which *p* attempts to communicate), or to detect the failure of the communication medium between *p* and *q* can be exploited by the application designer to make the application tolerate failures of processes and communication media. Consider, for example, an application that consists of a process *p* and two identical processes *q* and *r*. Assume that during some execution of this application, process *p* executes an alert primitive to receive a message containing some information from process *q*. Assume also that during this execution process *q* or the communication medium between *p* and *q* fails before *p* receives this message. Because the receive primitive is alert, execution of the receive primitive by *p* is aborted and process *p* is informed of the failure. Taking into account that this situation can arise, the application designer can design process *p* such that when this situation arises, process *p* obtains the same information from process *r*. In other words, the application is designed to reconfigure and let process *p* use process *r* instead of process *q* when process *q* or the communication medium between *p* and *q* fails.

Alert communications primitives differ from external failure detectors[10], system diagnosis[1], and other uses of heartbeat protocols such as reaching agreement[3], and mobile computing[7] in that the primitives themselves, used to send and receive messages, are capable of detecting failures. These other approaches can be used, however, to make communications primitives alert.

Given this useful role that alert communication primitives can play in reliable distributed applications over a TCP/IP network, it is reasonable to ask the following question: Are the communication primitives over TCP (the protocol for reliable message transmission in the Internet) alert? The next section firmly defines the concept of an alert communication primitive and argues that the send primitive over TCP is alert and the receive primitive over TCP is not. This section also presents three alternative receive primitives that are alert; these three primitives are called C-receive, D-receive, and E-receive. Implementations of these three primitives as functions in the C programming language are discussed in the next three sections. Subsequently, we compare the performance of these three primitives.

2 Alert Communication Primitives

A communication primitive is a send or receive primitive that can be executed by a process to send or receive a message from another process. Each sent message is transmitted through a communication medium between the sending process and the receiving process.

We make the following two assumptions concerning the failure of processes and communication media. First, once a process fails, then this process can never again send or receive any message. Thus, any message that is sent to a failed process will then be lost. Second, once a communication medium fails, then any message transmitted through this medium will also be lost. Note that a communication medium that has not failed can still lose some of the messages transmitted through it. However, any message that is transmitted repeatedly through such a medium will eventually reach the destination process of the message.

A communication primitive is called alert iff the following two conditions hold.

- *Persistence:* If a process *p* starts to execute this primitive to communicate with another process *q*, and if neither *q* nor the communication medium between *p* and *q* fails during the execution, then *p* successfully completes the execution of the primitive.

- *Perceptiveness:* If a process *p* starts to execute this primitive to communicate with another process *q*, and if *q* or the communication medium between *p* and *q* fails during the execution, then *p* detects the failure and aborts execution of the primitive.

An example of an alert communication primitive is the send primitive over TCP. Consider the case where an application process *p* executes a TCP primitive to send a message *m* to an application process *q* and assume that *p* and *q* execute above TCP processes *tp* and *tq*, respectively. This send primitive satisfies the two conditions of persistence and perceptiveness as follows.

First, if neither *q* nor the communication medium between *p* and *q* has failed, then the TCP process *tp* keeps on sending message *m* to the TCP process *tq* until *tp* receives an *ack* message from *tq*, acknowledging the reception of *m*. Thus, this send primitive satisfies the persistence condition.

Second, if process *q* fails, the TCP process *tq* informs *tp*, which informs process *p* of the failure. On the other hand, if the communication medium between *p* and *q* fails, then the TCP process *tp* will not receive an acknowledgment from the TCP process *tq*. Process *tp* concludes (after trying and failing for about 9 minutes) that a failure has occurred and informs process *p* of its conclusion. Thus, this send primitive satisfies the perceptiveness condition.

Unfortunately, the receive primitive over TCP is not alert. Consider the case where an application process *p* waits to receive a message from another application process *q*. In this case, if the communication medium between *p* and *q* fails while *p* is waiting to receive the message, then *p* may not be able to detect the medium failure and may continue to wait indefinitely. (On the other hand, if process *q* fails while *p* is waiting, then the TCP process *tq* detects this failure and sends a *finish* message to the TCP process *tp*, which in turn informs process *p* of the failure.)

At first glance, a simple modification seems capable of making the socket receive primitive alert. Consider the case where each receive primitive is augmented by a timeout that expires after *T* milliseconds, for some chosen *T*. Thus, if a process *p* waits to receive a message from a process *q* for *T* milliseconds without receiving a message, then *p* concludes that process *q* or the communication medium between *p* and *q* has failed and aborts execution of the receive primitive, as required by the perceptiveness condition.

The problem of this feature is that in general there is no acceptable value for *T*. For example, while process *p* is waiting for *T* milliseconds to receive a message *m* from process *q*, process *q* can be busy communicating with a third process and have no time to send message *m* to process *p*. In this case, the conclusion of *p* that *q* or the communication medium between *p* and *q* has failed, is wrong, and *p* should not have aborted its execution of the receive primitive. In other words, the receive primitive with the added time-out feature does not satisfy the persistence condition of an alert primitive.

In order to make the socket receive primitive over TCP alert, TCP can be modified as follows: Whenever a process *p* waits to receive a message *m* from a process *q*, the TCP process *tp* underneath *p* starts to send *beat* messages to the TCP process *tq* underneath *q*. (It is important that beat messages only be sent while process *p* is waiting to receive message *m*, since otherwise *p* may be erroneously informed of a failure when it will not wait for a message from process *q* in the future.) For each received beat message, process *tq* sends back an ack message to process *tp* if process *q* remains up. If both process *q* and the communication medium between *p* and *q* remain up, the exchange of beat and ack messages between *tp* and *tq* continues until process *p* receives message *m* from process *q*. On the other hand, if process *q* or the communication medium between *p* and *q* fails, then process *tp* can detect the failure (and inform process *p* of this failure) as follows. First, if *q* has failed, *tq* sends back a *reset* message for each received beat message. If *tp* receives a reset message then *tp* concludes that process *q* has failed. Second, if *tp* does not receive any ack or reset messages, then *tp* concludes that the communication medium between *p* and *q* has failed.

Because modifying TCP is a non-trivial task, it is important to find methods to make the socket receive primitive over TCP alert without modifying TCP. In other words, new socket receive primitives that are alert need to be defined over TCP. In this paper, we present three such primitives: C-receive, D-receive, and E-receive.

2.1 C-receive

When a process *p* invokes a C-receive primitive to receive a message *m* from a process *q*, the invoked primitive turns on the keep-alive timer (defined in Reference [2] and discussed in Reference [9]) in the TCP process *tp* underneath *p*, waits to receive *m* from *q* and then turns off the keep-alive timer. While the keep-alive timer *tp* is turned on, process *tp* periodically sends beat messages to the TCP process *tq* underneath process *q*, and receives ack messages acknowledging the reception of the beat messages. If process *tp* does not receive the ack messages for some time (an indication that the communication medium between *tp* and *tq* has failed) or if process *tp* receives one or more reset messages from process *tq* (an indication that process *q* has failed), then process *tp* informs process *p* of the failure. When this happens, process *p* abandons its wait for message *m*.

2.2 D-receive

When a process *p* invokes a D-receive primitive to receive a message *m* from a process *q*, the invoked primitive periodically sends beat messages to the TCP discard service[6] in the computer where process *q* is located until *p* receives *m* from *q*. The discard service discards all the (beat) messages sent to it. However, the beat messages are sent as TCP messages from the TCP process *tp* underneath process *p* to the TCP process *tq* underneath process *q*. Thus, reception of these beat messages is acknowledged automatically by sending ack messages from *tq* to *tp*. If the TCP process *tp* fails to receive an ack message for some sent beat message, *tp* concludes that the communication medium between *tp* and *tq* has failed and informs process *p*, which then abandons its wait for message *m*.

2.3 E-receive

When a process *p* invokes an E-receive primitive to receive a message *m* from a process *q*, the invoked primitive periodically sends beat messages to the UDP echo service[5] *eq* in the computer where process *q* resides. These beat messages are sent as UDP messages from the UDP process underneath process *p* to the echo service *eq* and so their receptions are not automatically acknowledged. Because an echo service sends back each received message to the process that generated the message, service *eq* sends back each received beat message to process *p*. If the E-receive primitive sends several beat messages to service *eq* and does not receive them back for some time, the E-receive primitive concludes that the communication medium between *p* and *q* has failed and abandons its wait for message *m*. (To be exact, this conclusion can be wrong! For example, if the UDP process in the computer where process *q* resides has failed, or if service *eq* has failed, then the E-receive primitive will not receive back any sent beat messages. However, the probability of this happening is very small and we can ignore this possibility for all practical purposes.)

In the following sections, we discuss how to implement the three primitives C-receive, D-receive, and E-receive in some detail.

3 Implementation of the C-receive Primitive

A C-receive primitive can be implemented using the following `Crecv` function.

```
int Crecv (int data, char *buf, size_t bufsize)
{
  int on = 1;
  int off = 0;
  int rc;

  /* Activate the TCP keepalive timer */
  if (setsockopt(data, SOL_SOCKET, SO_KEEPALIVE,
                 &on, sizeof(on))) {
    return(-1);
  }

  /* Read data from connection */
  rc = read(data, buf, bufsize);

  /* Deactivate the TCP keepalive timer */
  if (setsockopt(data, SOL_SOCKET, SO_KEEPALIVE,
                 &off, sizeof(off))) {
    return(-1);
  }

  return(rc);
}
```

First, this `Crecv` function turns on the keep-alive timer in TCP. Then, it waits until it receives the expected data from the remote site before it turns off the keep-alive timer. While the keep-alive timer is on, the TCP process in the local site periodically resends to the TCP process in the remote site the last byte that has been sent earlier to the remote site and has been acknowledged earlier by the remote site. When the TCP process in the remote site receives this byte, it merely sends an acknowledgment to the TCP process in the local site. When the TCP process in the local site receives the acknowledgment, it recognizes that the communications medium between the local and remote sites has not failed and does nothing.

If the TCP process in the local site fails to receive an acknowledgment for some resent byte, it recognizes that the communication medium between the local and remote sites has failed and signals the application that started the current execution of the `Crecv` function. This signal causes the `Crecv` function to terminate abruptly and abandon its wait for the expected data from the remote site.

4 Implementation of the D-receive Primitive

A D-receive primitive can be implemented using the function `Drecv`, defined below.

The `Drecv` function starts a loop where it waits, using a `select` construct, either for the expected data to arrive from the remote site, or for some specified timer initialized to value *Tm* to expire. Thus, one of the following three events occur first.

- *Successful termination:* If the expected data arrives first from the remote site, the `Drecv` function passes the arrived data to the application, and terminates.
- *More waiting:* If the timer expires first, the `Drecv` function executes the following steps:
  1. The `Drecv` function first locates the address of the data source in the remote site, and replaces the port number in this address with the port number of the TCP discard service. The resulting address is that of the TCP discard service in the remote site.
  2. The `Drecv` function then checks whether a TCP connection has already been established with the discard service in the remote site, and establishes such a connection if none has already been established.
  3. The `Drecv` function sends a beat message over the established TCP connection (to the discard service in the remote site), and restarts the loop after setting the timer to its specified value *Tm*.
- *Failure detection:* If the TCP process underneath the `Drecv` function does not receive an acknowledgment for the last sent beat message for some time, the TCP process causes the `Drecv` function to terminate abruptly indicating a failure detection.

The timer value *Tm* is chosen by the application designer as a compromise between resource usage (specifically, the number of messages sent) and the delay before a failure is detected. For example, if *Tm* is chosen to be 1 minute, then beat messages will be sent and acknowledged every minute and the maximum detection delay will be that 1 minute plus the time that the TCP process takes to retry sending the beat message, or about 10 minutes.

The `Drecv` function is defined as follows. (For brevity, most of the error checking code has been removed from this function.)

```
int Drecv (int data, char *buf, size_t bufsize)
{
  struct timeval timeout;
  fd_set datafdset;
  struct sockaddr_in remote;
  int rlen = sizeof(struct sockaddr_in);
  static int discard = -1;
  static unsigned long raddr;
  char *msg = "beat";

  while (1) {
    /* Prepare to wait for data and initialize timer */
    timeout.tv_usec = 0;
    timeout.tv_sec = Tm;
    FD_ZERO(&datafdset); FD_SET(data, &datafdset);

    /* Wait for data to become readable or timer to expire */
    select(data + 1, &datafdset, NULL, NULL, &timeout);

    /* If data is readable, return the results of reading */
    if (FD_ISSET(data, &datafdset)) {
      return(read(data, buf, bufsize));
    }

    /* If timer expires, then do 1, 2, and 3 */

    /* 1. Identify the remote host */
    memset(&remote, 0, sizeof(remote));
    getpeername(data, &remote, &rlen);

    /* 2. If this is the first call to Drecv or the remote
          host is different from the last call, connect to
          the discard service on the remote host */
    remote.sin_port = htons(TCP_DISCARD);
    if (discard == -1 || remote.sin_addr.s_addr != raddr) {
      if (discard != -1) { close(discard); }
      discard = socket(PF_INET, SOCK_STREAM, 0);
      connect(discard, &remote, sizeof(remote));
      raddr = remote.sin_addr.s_addr;
    }

    /* 3. Send a beat message to the discard
          service on remote host */
    write(discard, msg, strlen(msg));
  }
}
```

Note that this `Drecv` function supports two optimizations. First, if the `Drecv` function is started and the expected data arrived within *Tm* seconds, then the `Drecv` function receives the data without overhead, just like the original TCP socket receive primitive. Second, the `Drecv` function establishes only one TCP connection with the discard service in each remote site. Once a TCP connection has been established with a remote site, each invocation of the `Drecv` function with that site merely uses the established connection.

To illustrate the execution of this `Drecv` function, we set up an experiment where an HTTP client communicates with an HTTP server using a version of this function where $Tm = 60$ seconds. The HTTP client is executed on a Sun SPARCstation, called bark, running Solaris 2.5. The HTTP server is executed on an Intel PentiumPro machine, called capuchon, running Linux. (Our choice of HTTP is intended for illustration only, to be simple to implement and simple to understand. We do not intend to suggest that these primitives should be used with HTTP.)

The communication between the client and the server is carried over a connection between TCP port 63433 in the client machine, bark, and TCP port 5000 in the server machine, capuchon. A tcpdump trace of this communication is as follows.

```
 1    0.00         bark.63433  >  capuchon.5000     :  S(0)
 2    0.00      capuchon.5000  >  bark.63433        :  S(0) ack
 3    0.00         bark.63433  >  capuchon.5000     :  . ack
 4    0.00         bark.63433  >  capuchon.5000     :  P 1:18(17) ack
 5    0.02      capuchon.5000  >  bark.63433        :  . ack
 6   60.00         bark.63434  >  capuchon.discard  :  S(0)
 7   60.01   capuchon.discard  >  bark.63434        :  S(0) ack
 8   60.01         bark.63434  >  capuchon.discard  :  . ack
 9   60.01         bark.63434  >  capuchon.discard  :  P 1:5(4) ack
10   60.03   capuchon.discard  >  bark.63434        :  . ack
11  120.00         bark.63434  >  capuchon.discard  :  P 5:9(4) ack
12  121.49         bark.63434  >  capuchon.discard  :  P 5:9(4) ack
13  124.47         bark.63434  >  capuchon.discard  :  P 5:9(4) ack
14  130.44         bark.63434  >  capuchon.discard  :  P 5:9(4) ack
15  142.37         bark.63434  >  capuchon.discard  :  P 5:9(4) ack
16  166.22         bark.63434  >  capuchon.discard  :  P 5:9(4) ack
17  213.92         bark.63434  >  capuchon.discard  :   . 5:9(4) ack
18  270.17         bark.63434  >  capuchon.discard  :   . 5:9(4) ack
19  326.43         bark.63434  >  capuchon.discard  :   . 5:9(4) ack
20  382.68         bark.63434  >  capuchon.discard  :   . 5:9(4) ack
21  438.93         bark.63434  >  capuchon.discard  :   . 5:9(4) ack
22  495.18         bark.63434  >  capuchon.discard  :   . 5:9(4) ack
23  551.43         bark.63434  >  capuchon.discard  :   . 5:9(4) ack
24  660.02         bark.63433  >  capuchon.5000     :  F 18:18(0) ack
25  661.49         bark.63433  >  capuchon.5000     :  F 18:18(0) ack
26  664.42         bark.63433  >  capuchon.5000     :  F 18:18(0) ack
27  670.27         bark.63433  >  capuchon.5000     :  F 18:18(0) ack
28  681.97         bark.63433  >  capuchon.5000     :  F 18:18(0) ack
```

Each line in this trace defines one message that is sent between bark and capuchon. Lines 1–3 define the three way handshake for establishing a TCP connection between port 63433 in bark and port 5000 in capuchon. Line 4 defines a 7 byte request message sent from the client to the server. After sending this request, the client invokes the `Drecv` function to wait for the expected reply. This reply, however, will not arrive because the server is modified such that it does not reply to the request. Line 5 defines an acknowledgment sent from capuchon to bark acknowledging the reception of the request. At time 60 seconds, the `Drecv` function in bark notices that the reply has not arrived and establishes a TCP connection with the discard server in capuchon, as indicated by lines 6–8. The `Drecv` function then sends a 4 byte beat message to the discard server, as indicated by line 9, and the sent beat message is acknowledged, as indicated by line 10.

The second beat message is to be sent at time 120 seconds. However, some time between 60 seconds and 120 seconds, we remove the server machine capuchon from the network in order to simulate some failure in the communication medium between bark and capuchon. At time 120 seconds, the `Drecv` function sends the second beat message, as indicated by line 11. Because this beat message is not acknowledged, bark times out (using its exponential back off mechanism) and resends this message over and over, as indicated by lines 12–23. After 9 minutes, bark concludes correctly that the communication medium between bark and capuchon has failed and starts to send a finish message to remove the established TCP connection between bark and capuchon, as indicated by lines 24–28.

5 Implementation of the E-receive Primitive

An E-receive primitive can be implemented using the function `Erecv` defined below. When this `Erecv` function is invoked (to wait for some data from a remote site), the function periodically sends UDP beat messages to the echo service in the remote site. The echo service receives the beat messages and sends them back to the `Erecv` function in the local site. As long as the `Erecv` function keeps on receiving the beat messages it has sent earlier, the `Erecv` function concludes that the communication medium between the local and remote sites has not failed and continues to wait for the expected data from the remote site.

The length T of the time period between sending two successive beat messages by the `Erecv` function changes over time, according to the accelerated heartbeat protocol[4], as follows. If the `Erecv` function sends a beat message and later receives it back, then T is assigned its maximum value *Tmax*. If the `Erecv` function sends a beat message but does not receive it back, then the value $T/2$ is compared with a value *Tmin* that is an upper bound on the round trip delay between the local and remote sites. If $T/2$ is at least *Tmin*, then T is assigned the value $T/2$. Otherwise, the `Erecv` function concludes that the communication medium between the local and the remote sites has failed, abandons its wait for the expected data from the remote site, and abruptly terminates.

As described in Reference [4], the values *Tmax* and *Tmin* are chosen by the application designer as a compromise between the number of sent beat messages, the delay in detecting failures, and the probability of erroneous failure detection (due to message loss). For example, by choosing *Tmax* to be 200 seconds and *Tmin* to be 2 seconds, the `Erecv` function sends 7 beat messages before it detects a failure and informs process p of the failure. Thus, the delay in detecting failures is 10 minutes, and the probability of erroneous failure is around 1/6000 for connections lasting an hour where the probability that a message is lost is 0.1.

The `Erecv` function starts a loop where it waits, using a `select` construct, either for the expected data to arrive from the remote site, or for some specified timer to expire. (The value T of this timer is initialized to *Tmax*.) Thus, one of the following two events occurs first:

- *Successful termination:* If the expected data arrives first from the remote site, the `Erecv` function passes the arrived data to the application and terminates.

- *More waiting or failure detection:* If the timer expires first, the `Erecv` function checks whether a UDP socket for communicating with the echo service in the remote site has been created earlier. There are two cases to consider.

1. If this socket has not been created earlier, then the `Erecv` function creates such a socket, sends a beat message over this socket to the echo service in the remote site, sets the value of the timer to $Tmax/2$ (because it treats the first timeout period *Tmax* as if it ended by the loss of a beat message), and restarts the loop.
2. If this socket has been created earlier, then the `Erecv` function checks whether the last sent beat message has returned to this socket. If so, the `Erecv` function sends the next beat message over this socket, sets the value of the timer to *Tmax*, and restarts the loop. Otherwise, the `Erecv` function compares the value $T/2$ with *Tmin*. If $T/2$ is at least *Tmin*, the `Erecv` function sends the next beat message over the socket, sets the value of the timer to $T/2$, and restarts the loop. Otherwise, the `Erecv` function concludes that the communication medium between the local and remote sites has failed and terminates.

The `Erecv` function is defined as follows.

```
int Erecv (int data, char *buf, size_t bufsize)
{
  int T;
  struct timeval timeout;
  fd_set datafdset;
  int echo = -1;
  struct sockaddr_in remote;
  int rlen = sizeof(struct sockaddr_in);
  fd_set echofdset;
  struct timeval poll = {0,0};
  char rply[MSGLEN];
  char *msg = "beat";

  /* Initialize timer */
  T = Tmax;
  timeout.tv_usec = 0;
  timeout.tv_sec = T;
  while (1) {

    /* Wait for data to become readable or for timer to expire */
    FD_ZERO(&datafdset); FD_SET(data, &datafdset);
    select(data + 1, &datafdset, NULL, NULL, &timeout);

    /* If data is readable, close the UDP socket if
       needed and return the results of reading */
    if (FD_ISSET(data, &datafdset)) {
      if (echo != -1) { close(echo); }
      return(read(data, buf, bufsize));
    }

    /* If no UDP socket has been set up, get the
       address of the UDP echo service on remote
       host and create socket to send messages to
       it */
    if (echo == -1) {
      memset(&remote, 0, sizeof(remote));
      getpeername(data, &remote, &rlen);
      remote.sin_port = htons(UDP_ECHO);
      echo = socket(PF_INET, SOCK_DGRAM, 0);
      connect(echo, &remote, sizeof(remote));
    }

    /* Check whether reply to last beat message is available */
    FD_ZERO(&echofdset); FD_SET(echo, &echofdset);
```

```
        select (echo + 1, & echofdset , NULL, NULL, &poll);
        if ( FD_ISSET(echo, &echofdset)) {
          /* If so , set the delay to Tmax */
          T = Tmax;
          timeout. tv_usec = 0;
          timeout. tv_sec = T;
          read (echo , rply , MSGLEN);
        } else {
          /* If not , reduce the delay by half */
          T = T / 2;

          /* If the new delay is below Tmin, abort */
          if (T < Tmin) { errno = EIO; return (−1); }

          timeout. tv_usec = 0;
          timeout. tv_sec = T;
        }

        /* If the new delay is above Tmin, send a beat message */
        write (echo , msg, strlen (msg));
      }
    }
```

To illustrate the execution of this `Erecv` function, we set up an experiment with an HTTP server using a version of this function, where $Tmax = 200$ seconds and $Tmin = 2$ seconds. The client and the server are executed on the two machines bark and capuchon, respectively, mentioned earlier. A tcpdump trace of the communication between the client and the server is as follows.

```
 1    0.00      bark.38584  > capuchon.5000  :  S(0)
 2    0.00   capuchon.5000  > bark.38584     :  S(0) ack
 3    0.00      bark.38584  > capuchon.5000  :  . ack
 4    0.00      bark.38584  > capuchon.5000  :  P 1:18(17) ack
 5    0.02   capuchon.5000  > bark.38584     :  . ack
 6  200.01      bark.45929  > capuchon.echo  :  udp 4
 7  200.01   capuchon.echo  > bark.45929     :  udp 4
 8  300.01      bark.45929  > capuchon.echo  :  udp 4
 9  300.01   capuchon.echo  > bark.45929     :  udp 4
10  500.02      bark.45929  > capuchon.echo  :  udp 4
11  700.21      bark.45929  > capuchon.echo  :  udp 4
12  800.23      bark.45929  > capuchon.echo  :  udp 4
13  850.23      bark.45929  > capuchon.echo  :  udp 4
14  875.23      bark.45929  > capuchon.echo  :  udp 4
15  887.23      bark.45929  > capuchon.echo  :  udp 4
16  893.23      bark.45929  > capuchon.echo  :  udp 4
17  896.28      bark.38584  > capuchon.5000  :  F 18:18(0) ack
18  897.71      bark.38584  > capuchon.5000  :  F 18:18(0) ack
19  900.58      bark.38584  > capuchon.5000  :  F 18:18(0) ack
20  906.32      bark.38584  > capuchon.5000  :  F 18:18(0) ack
```

Lines 1–5 in this trace are similar to lines 1–5 in the previous trace. After sending the request in line 4, the client invokes the `Erecv` function to wait for the expected reply that will not arrive because the server is modified such that it does not reply to the request. At time 200 seconds, the `Erecv` function in bark notices that the reply has not arrived, and sends the first beat message and makes $T = 100$ seconds, as indicated by line 6. The echo service in capuchon receives this beat message and sends it back to the `Erecv` function, as indicated by line 7. At time 300 seconds, the `Erecv` function notices that the reply still has not arrived but the last beat message has been returned. Thus, the `Erecv` function sends the second beat message and makes $T = 200$ seconds, as indicated by line 8. The echo service receives the beat message and sends it back to the `Erecv` function, as indicated by line 9.

The next beat message is to be sent at time 500 seconds. Some time between 300 seconds and 500 seconds, the server machine capuchon is removed from the network to simulate a failure in the communication medium between the two machines. Therefore, all sent messages in the trace after 500 seconds are from bark to capuchon. Lines 10–16 indicate that bark sends beat messages at accelerated rates (because it does not receive back any of these beat messages), and halves the value of T each time. Thus, the value of T at line 16 is 1.5 seconds, which is less than the value $Tmin$ of 2 seconds, and bark concludes correctly that the communications medium between the two machines has failed. In lines 17–20, bark attempts to send a finish message to remove the established TCP connection between bark and capuchon.

6 Comparing the Three Primitives

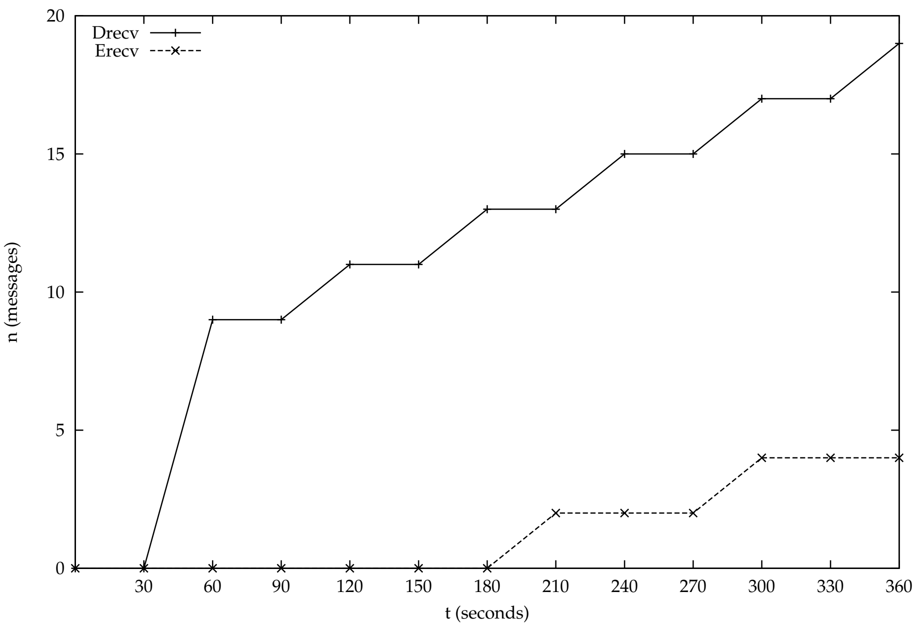
In this section, we compare the three alert receive primitives, C-receive, D-receive, and E-receive, discussed in the previous sections. We start by observing that the C-receive primitive is not satisfactory, for three reasons. First, this primitive is based on an optional feature of TCP, namely the keep-alive timer. Second, the C-receive primitive uses the keep-alive timer in a way that was not specified in Reference [2]. In particular, it is not clear from Reference [2] that turning the keep-alive timer on and off repeatedly as required when a sequence of C-receive primitives is to be executed, is an acceptable use of the keep-alive timer. Third, when an application invokes a C-receive primitive, the application cannot dictate the rate at which this invoked primitive generates and sends beat messages. (For some applications, this rate should be high in order to reduce the delay for detecting failures, and for other applications, this rate should be low in order to reduce the overhead.) For these reasons, we ignore the C-receive primitive and focus on comparing the performance of the D-receive and E-receive primitives.

To compare the performance of D-receive and E-receive primitives, we carried out an experiment between a HTTP client and HTTP server running on the two machines bark and capuchon mentioned earlier. In this experiment, the client sends a request message to the server, then waits to receive a reply message from the server. After receiving the request message, the server waits for t seconds before sending back the reply message. The experiment is executed several times. In some executions, the client waits to receive the reply message by invoking a D-receive primitive. In other executions, the client waits to receive the reply message by invoking an E-receive primitive. The parameters of the D-receive and E-receive primitives, used in the client, are chosen such that the two primitives have the same delay in detecting failures. In particular, Tm of the D-receive primitive is chosen to be 60 seconds, and $Tmax$ and $Tmin$ of the E-receive primitive are chosen to be 200 seconds and 2 seconds, respectively.

For every execution of the experiment, we measured the waiting time t and the number n of overhead messages (mostly beat and ack messages) that were exchanged between the client and the server while the client was waiting to receive the reply message. The measured results are shown in Figure 1.

From Figure 1, we conclude that the relationships between the number n of overhead messages and the waiting time t in the two cases of the D-receive and E-receive

Figure 1: Waiting time and overhead messages



primitives are as follows.

$$\text{D-receive:} \quad n = 7 + 2\left\lfloor \frac{t}{Tm} \right\rfloor$$

$$\text{E-receive:} \quad n = 2 + 2\left\lfloor \frac{t}{Tmax} \right\rfloor$$

Note that t is measured in seconds, $Tm = 60$ seconds and $Tmax = 200$ seconds.

Because Tm is less than $Tmax$, the number of overhead messages in the case of a D-receive primitive is larger than those in the case of an E-receive primitive. Thus, E-receive primitives are more efficient than D-receive primitives.

7 Concluding Remarks

In this paper, we introduced the concept of an alert communication primitive, and argued that the send primitive over TCP is alert but the receive primitive over TCP is not. Then, we proposed three alternative alert receiving primitives: C-receive, D-receive, and E-receive. We provided implementations of the three primitives in C as `Crecv`, `Drecv`, and `Erecv`. These three functions are designed to be used transparently in place of the TCP receive primitive. We compared the performance of these three primitives and reached the conclusion that E-receive is more flexible than C-receive, and is more efficient than D-receive.

Applications that can benefit from the use of alert communication primitives include those that have reliability constraints, such as remote file system clients, which should not wait indefinitely for the response to a request. Also, applications such as complex interactive applications, which use significant resources, would benefit since it would allow them to free the resources rather than hold them indefinitely. The use of alert primitives as described in this paper allows these benefits to be realized without difficult modifications to existing programs.

Note that in our implementation of the D-receive and E-receive primitives, if a primitive is invoked to receive a message *m*, then this invocation does not cause any beat messages to be sent for some time period in the hope that message *m* will be received within this time period. Thus, if messages are to be received within reasonable time periods after invoking the receive primitives, then using D-receive or E-receive primitives will not cause any (overhead) messages to be sent.

Our implementation of the D-receive and E-receive primitives are based on the BSD socket interface, but only use the standard behavior of TCP and UDP. As a result, it is possible to port them to other implementations of TCP. For example, when using the System V Transport Layer Interface[8], the D-receive and E-receive primitives can be implemented by a loop using a `poll` construct. Additionally, it is possible to generalize the D-receive and E-receive primitives to other primitives that wait to receive messages from other processes, such as the BSD `select` function.

References

[1] M. Barborak, M. Malek, and A. Hahbura. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2), June 1993.

[2] R. Braden. RFC 1122: Requirements for Internet hosts — communication layers, October 1989.

[3] F. Cristian. Reaching agreement on processor group membership. *Distributed Computing*, 4, 1991.

[4] Mohamed G. Gouda and Tommy M. McGuire. The accelerated heartbeat protocols. In *18th International Conference on Distributed Computing Systems*. IEEE Computer Society, May 1998.

[5] J. Postel. RFC 862: Echo protocol, May 1983.

[6] J. Postel. RFC 863: Discard protocol, May 1983.

[7] Dhiraj K. Pradhan, P. Krishna, and N. Vaidya. Recoverable mobile environment design and trade-off analysis. In *FTCS '96*, June 1996.

[8] Stephen A. Rago. *UNIX System V Network Programming*. Addison Wesley, 1993.

[9] W. Richard Stevens. *TCP/IP Illustrated*, volume 1. Addison Wesley, 1994.

[10] Werner Vogels. World wide failures. In *ACM SIGOPS 1996 European Workshop*. Association for Computing Machinery, September 1996.