Message-Efficient Uniform Timed Reliable Broadcast

Yu Ma and Scott D. Stoller

21 September 1998

1. Introduction

In distributed database systems, atomic commitment protocols ensure that transactions leave the database in a consistent state even if failures occur during transactions. As shown by Babaoglu and Toueg, the heart of the atomic commitment problem is equivalent to uniform timed reliable broadcast (UTRB), which is a broadcast primitive that provides the following guarantees [Babaoglu and Toueg]:

B1 (Validity): If a correct process broadcasts a message *m*, then all correct processes eventually deliver *m*.

B2 (Integrity): For any message *m*, each process delivers *m* at most once, and only if some process actually broadcasts *m*.

B3 (Δ_b-Timeliness): There exists a known constant Δ_b such that if the broadcast of *m* is initiated at real-time *t*, no process delivers *m* after real-time $t+\Delta_b$.

B4 (Uniform Agreement): If any process (correct or not) delivers a message *m*, then all correct processes eventually deliver *m*.

We consider two complexity metrics for UTRB algorithms: **time** and **number of messages.** Let N be the number of processes. Assume that communication is FIFO and reliable, and each message is received within δ time units (as measured in real-time) after being sent. Local overhead of communication is captured by a parameter τ: after a process sends a set of at most N-1 messages to different processes, τ time units elapse before more messages can be sent.

It is easy to devise a time-optimal UTRB protocol, e.g., UTRB1 of [Babaoglu and Toueg], which works as follows. Each process relays every message it receives to all other processes before delivering the message; thus, if any process delivers a message, then that message must have been sent to all processes. Let *f* denote the total number of processes that crash during an execution of the protocol, then the worst-case time complexity of UTRB1 is only $(f+1)\delta$, but the worst-case number of messages is $(N-1)^2$.

What about a message-optimal algorithm then? The most message-efficient algorithm in the literature is UTRB2 [Babaoglu and Toueg], which uses three types of messages: MSG announces a broadcast, DLV causes a delivery, and REQ requests help. The initial broadcaster constructs a list of processes called cohorts that will cooperate in performing the broadcast. The first process on this list is the broadcaster itself. To tolerate the failure up to *F* processes, the list contains *F+1* distinct process name. This list, along with the index of the current cohort is included in MSG and REQ messages. Processes that received MSG from the broadcaster but didn't receive DLV after waiting for an additional $\delta+\tau$ time units would time out and send REQ to a cohort for help, and they wait for $2\delta+\tau$ before sending REQ to the next cohort. This procedure repeats until either a DLV arrives and causes delivery, or there are no more cohorts to ask for help. On receiving REQ, a cohort send MSG and DLV in the same way as the initial broadcaster to all other processes. In case of *f* failures $(F>f>0)$ in an execution of the protocol, the worst-case time complexity of UTRB2 is $(f+1)(2\delta+\tau)$ and the worst-case number of messages is $2(f+1)(N-1)$.

This paper describes a new UTRB protocol, called UTRB4, which has worst-case message complexity *2(N-1)+ f(f-1)/2*, which is asymptotically better than UTRB2. We conjecture that UTRB4 is asymptotically message-optimal. However, the worst-case time complexity of UTRB4 is exponential in N. For example, when f = 1 and N > 3, the worst case message and time complexities of UTRB4 are *2(N -1)* and $2\delta + \tau + 2^{(N-1)}\delta + 2^{(N-4)}\tau$ respectively compared to *4(N-1)* and *2(2*$\delta + \tau$*)* for UTRB2. We are also studying the trade-off between time and number of messages. We conjecture that no algorithm can be optimal with respect to both metrics.

2. Description of UTRB4

Figure1 is the main structure of UTRB4.

```
Pi :: initially bcasting := true;
   do
      [] bcasting→ bcast(msg); bcasting := false;
      [] ¬ bcasting→ msg.data := new message; msg.src := i; bcasting := true
                     or skip
      [] S ? <MSG, m> → msg_rcvd ⊕ = m;
      //By MP4, MSG cannot have been received before.
      //S may be the broadcaster or the helper. By P2, if it is S's turn to help with the
      //MSG, lower-ranked processes must have crashed.
                              //The next process's rank to ask for help is stored in r[m];
                              r[m] = rank(S, m.src) + 1;
                              //k is the rank difference of the two processes that are
                              //communicating.
                              k = rank(i,m.src) – rank(S,m.src);
                              create_timer(m);
                              set_timer(m, T_m(k));                                  (1)
      [] S?<DLV, m> → deliver(m); dlvd ⊕ =m;
                           delete_timer(m);

      [] Timer[m] → if  r[m] = rank(i, m.src)  then
                              help_i(m, i)
                     else
                              k = rank(i, m.src) – r[m];
                              rank^-1(r[m], m.src) ! <REQ, m>;
                              set_timer(m, T_r(k));                                  (2)
                              r[m]++;
                     fi
      [] S?<REQ, m>→ if m ∉ helped then
                              help_i(m,S);
                        fi
   od
```

Figure 1. UTRB4: A Message-Efficient UTRB Algorithm

To reduce the message complexity from UTRB2, we arrange for a process to assume the role of the initial broadcaster only in case of certain failures.

The algorithm also uses three types of messages: MSG announces the message, DLV causes a delivery, and REQ is used to request help. Every process has its own process identifier (PID). Let N be the number of processes, then PID = [0, N-1]. Each process is also assigned a unique rank according to its PID and the PID of the initial broadcaster. Function *rank* converts a PID into a rank. For all i, s ∈ PID, *rank* (i, s) = (i - s) mod N is the rank of the process with PID i with respect to the sender s. For any r, $rank^{-1}$(r, s) = (r + s) mod N is the PID of the process of rank r with respect to sender s. Each message m indicates the PID of the sender with a special field in the message, m.src. Statement *create_timer(m)* creates a timer *m*. *set_timer(m, t)* sets timer *m* to length *t* of time. *delete_timer(m)* deletes timer *m* regardless of whether *m* has timed out yet.

Figure 2 is the broadcast procedure called in the algorithm. It indicates the order that MSG and DLV messages are sent.

```
procedure bcast(m)
        for d := N-1 downto 1
                        rank^-1(d, m.src)!<MSG,m>;
        for d := 1 to N-1
                        rank^-1(d, m.src)!<DLV, m>;
        deliver(m);
```

Figure 2. Broadcast Procedure of UTRB4

If the broadcaster fails, a helper helps processes deliver the message by sending MSG and DLV to appropriate processes. Both the initial broadcaster and helpers send MSG to processes in decreasing order by rank, and DLV in increasing order by rank.

If some process *p* received MSG from a process ranked *q* but timed out while waiting for DLV, by careful selection of time-outs, *p* knows that the sender of MSG must have crashed and sends REQ for help to the process with rank *q+1*. If this helper is correct, on receiving the REQ, it helps all correct processes deliver the message. Otherwise, *p* again times out waiting for DLV and sends REQ to the next higher-ranked process. If all processes ranked lower than *p* crashed, *p* eventually times out after having sent a REQ message to the process with rank *rank(p,m.src)-1*. In this case, *p* becomes a helper.

Figure 3 shows the help procedure described above, which is very important in reducing the message complexity of the algorithm.

3. Selecting Time-Outs for UTRB4

We can obtain the following properties by selecting time-outs carefully, in order to use as few REQ messages as possible and to avoid sending redundant MSG and DLV messages as much as possible. The order that MSG, DLV and REQ are sent, namely, in order of decreasing, increasing, and increasing rank, respectively, is also essential.

Without loss of generality, we assume in the following discussion that process 0 is the broadcaster; thus the rank of each process is the same as its PID, so we use them interchangeably.

```
//j asked i to help with message m
procedure help_i(m, j)
// It's clear that rank(j, m.src) ≥ rank(i, m.src).
        helped ⊕ = m;
        if m ∈ msg_rcvd  then
        // By R1 and R2, if j ≠ i, then rank (j, m.src) > rank (i, m.src) and i received //DLV
        already, otherwise i would time out and ask for help earlier than j.

        //0 to rank (i, m.src)-1: By P2, processes in this range have crashed.

        //rank (i, m.src)+1 to rank (j, m.src): This range is empty when j = i. Consider the
        //case where j ≠ i, then i received DLV already. By P5, MSG must have been sent
        //to all correct processes ranked in this range. By R1 and R2, correct processes in
        //this range should time out and ask for help earlier than j did. So all processes
        //ranked in this range must have crashed.

        //max (rank(i, m.src)+1, rank(j, m.src)) to N-1: By P3 and R1,R2, processes in
        //this range either crashed or received MSG but did not time out on DLV yet;

                for d := max ( rank(i, m.src)+1, rank(j, m.src)) to N-1
                        rank^-1(d, m.src) ! <DLV, m>;
        else

        //N-1 downto rank(j, m.src)+1: By P3 and R1,R2, processes in this range either
        //crashed or received MSG but did not time out on DLV yet;

        //rank(j, m.src)-1 downto rank(i, m.src)+1: By P4 , processes ranked in this range
        //either crashed or didn't receive MSG;

        //rank(i, m.src)-1 downto 0: By P2, processes in this range have crashed.

                for d := rank(j, m.src)-1 downto  rank(i, m.src) +1                    (3)
                        rank^-1(d, m.src) ! <MSG, m>;
                msg_rcvd ⊕ = m;
                for d := rank(i, m.src)+1 to N-1
                        rank^-1(d, m.src) ! <DLV, m>;

        fi

        if m∉ msg_dlvd then
                dlvd⊕ = m;
                deliver(m);
        fi
```

Figure 3. Help Procedure of UTRB4

P1 All correct processes that receive MSG or REQ eventually deliver the message.
P2 If a process p starts to execute the help procedure, then all processes ranked lower than p have crashed.
P3 If a process p receives MSG, then MSG has been sent to all processes ranked higher than p.
P4 If a process p receives REQ from q, then q is ranked higher than p and if p didn't receive MSG from any process, then all processes ranked between p and q either crashed or didn't receive MSG.
P5 If a process p receives DLV, then MSG must have been sent to all correct processes, and all correct processes ranked lower than p have delivered or will deliver the message.

We give mutually recursive rules ***R1*** and ***R2*** that define time-outs T_m (used in line (1) of the algorithm) and T_r (used in line (2) of the algorithm) respectively. We prove below that this choice of time-outs ensures properties **P1-P5**.

R1.a If process p receives MSG from the broadcaster, the time p should wait before timing out waiting for DLV from the broadcaster or a helper and sending a REQ message is $T_m(p) = T_1(p) + T_2(p) + T_3(p) + T_4(p)$, where :

1. $T_1(p)$: time for all MSG's sent by the broadcaster to processes ranked lower than p to be received, i.e. time from when p received MSG from the broadcaster until all MSG's sent by the broadcaster to processes ranked lower than p have been received.
   If p = 1, then p is the last process the broadcaster sends MSG to, so $T_1(p) = 0$;
   Else $T_1(p) = \delta$;
2. $T_2(p)$: time from when processes ranked lower than p received MSG from the broadcaster until they have all timed out waiting for DLV from the broadcaster or a helper and sent some REQ's.
   If p = 1, then p should allow τ time units to elapse before the broadcaster sends the first DLV message and δ time units for that message to arrive, so $T_2(p) = \delta+\tau$;
   Else $T_2(p)$ is the maximum time-out of processes ranked lower than p after receiving MSG from the broadcaster, which by induction is $T_m(p-1)$.
3. $T_3(p)$: time from when processes ranked lower than p timed out waiting for DLV from the broadcaster or a helper and sent their first REQ's until one of those processes starts to provide help if it doesn't crash. In the worst case, one of these processes repeatedly sends a REQ and times out waiting for DLV and then finally starts to provide help itself.
   If p = 1 or 2, no process ranked lower than p needs to send any REQ's, so $T_3(p) = 0$;
   Else $T_3(p)$ is the maximum for processes ranked lower than p of the sum of the time-outs after the sending of each REQ, as discussed in ***R2,*** which by induction is $\Sigma_{j=1..p-2} T_r(p-1, j)$, where T_r is defined below in ***R2.***

4. $T_4(p)$: time from when process p-1 started to provide help itself (since this was the worst case for $T_3(p)$) until a DLV message arrives at process p. Note that when process p-1 started to provide help itself, it must have timed out waiting for DLV messages from all processes ranked lower than it, and MSG must have been sent to all processes ranked higher than it (because MSG is sent to processes in the decreasing order by rank). Thus, in the help procedure, DLV messages need to be sent only to processes ranked higher than p-1.

   If $p = 1$, $T_4(p) = 0$;

   Else $T_4(p) = \delta$.

R1.b If process p receives MSG from a helper q, the time p should wait before timing out waiting for DLV from q or another helper is $T_m(p,q)$, defined as follows. Since helpers send messages in the same order as the initial broadcaster, selecting time-outs $T_m(p, q)$ for p is the same as described above, except that now we should consider the "relative" rank of p to the helper q, which is reflected by the rank difference of p and q. Note that $q<p$. Thus $T_m(p, q) = T_1(p, q) + T_2(p, q) + T_3(p, q) + T_4(p, q)$, where:

1. $T_1(p, q)$: If $p-q = 1$, $T_1(p, q) = 0$; Else $T_1(p, q) = \delta$;
2. $T_2(p, q)$: If $p-q = 1$, $T_2(p, q) = \delta+\tau$; Else $T_2(p, q) = T_m(p-1, q)$;
3. $T_3(p, q)$: If $p-q = 1$ or 2, $T_3(p, q) = 0$;

   Else $T_3(p, q) = \Sigma_{j=q+1..p-2} T_r(p-1, j)$;
4. $T_4(p, q)$: If $p-q = 1$, $T_4(p, q) = 0$; Else $T_4(p, q) = \delta$.

Note that $T_m(p) = T_m(p, 0)$. Thus we can summarize ***R1.a*** and ***R1.b*** with the formulas below:

If $p-q = 1$, $T_m(p,q) = \delta+\tau$;

Else if $p-q = 2$, $T_m(p,q) = \delta+T_m(p-1, q)+ \delta = \delta+\delta+\tau+\delta$;

Else $T_m(p,q) = \delta+T_m(p-1, q)+\Sigma_{j=q+1..p-2} T_r(p-1, j)+\delta$.

R2. If process p sends REQ to process q, where $q<p$, the time $T_r(p, q)$ that p should wait before sending next REQ to q+1 includes:

1. Time δ for q to receive the REQ message, i.e. time from when p sent the REQ message until this message arrived at q.
2. Time $T'_r(p, q)$ from when q received REQ until some process q', where $q \leq q' < p$, sent DLV to p. It is possible that q crashes during the help procedure, so some process q' is involved. Recall that time-outs reflect the maximum waiting time needed. In the worst case, process q crashes immediately after sending MSG to process p-1; this causes the longest waiting time of all processes ranked between q and p on receiving MSG from q as discussed in ***R1,*** because $T_m(p,q)$ is an increasing function of p. So $q' = p-1$, and we have:

   If $p-q = 1$, no process is ranked between p and q, so $T'_r(p, q) = 0$;

   Else if $p-q = 2$, it takes δ time units for the MSG to arrive at $q' = p-1$, and $T_m(p-1, q)$ before q' timed out waiting for DLV from q and started to provide help itself, so $T'_r(p, q) = \delta + T_m(p-1, q) = \delta+\delta+\tau$;

   Else we also need to include the time q' needs to send all necessary REQ messages and time out waiting for DLV from each before it starts to

provide the help itself, in which case $T'_r(p, q) = \delta + T_m(p-1, q) + \Sigma_{j = q+1..p-2} T_r(p-1, j)$.

3. Time δ for DLV sent to p to arrive, i.e. time from when q' sent the DLV message to p until this message arrived at p.

Summarizing item 1-3, we have:

If p-q = 1, $T_r(p,q) = \delta+\delta$;

Else if p-q = 2, $T_r(p,q) = \delta+\delta+\delta+T_m(p-1, q) = \delta+\delta+\delta+\delta+\tau$;

Else $T_r(p,q) = \delta+\delta+\delta+T_m(p-1, q)+\Sigma_{j=q+1..p-2} T_r(p-1, j)$.

Now it's easy to get the relation between $T_m(p, q)$ and $T_r(p, q)$ as follows:

If p-q = 1, $T_r(p, q) = T_m(p, q) + \delta - \tau$;

Else $T_r(p, q) = T_m(p, q) + \delta$. (3)

These rules guarantee that at most one process is qualified to send the REQ message at any particular time. Thus by looking at the rank of the process that sent the REQ message, a helper is able to tell whether each correct process has received MSG and DLV or not. A helper thus can avoid sending unnecessary MSG and DLV to correct processes. This is important in reducing the message complexity.

3.1 Computation of Time-outs

Now we compute the general expression for recursively defined time-outs, based on ***R1*** and ***R2.*** For simplicity, let $T_m(k)$ and $T_r(k)$ represent $T_m(p, q)$ and $T_r(p, q)$ respectively, where k is the rank difference between p and q.

$T_m(k) = \delta+T_m(k-1)+ \Sigma_{j=1..k-2} T_r(j) +\delta$;

We use (3) to eliminate T_r.

$T_m(k) = 2\delta + T_m(k-1) + \Sigma_{j=1..k-2}(\delta+T_m(j))-\tau$

$T_m(k) = 2\delta+(k-2)\delta+ \Sigma_{j=1..k-1} T_m(j) - \tau$

$T_m(k) = k\delta + \Sigma_{j=1..k-1}T_m(j) - \tau$

$T_m(k-1) = (k-1)\delta + \Sigma_{j=1..k-2} T_m(j) - \tau$

$T_m(k) - T_m(k-1) = \delta + T_m(k-1)$

$T_m(k) - 2T_m(k-1) = \delta = T_m(k-1) - 2T_m(k-2)$

$T_m(k) - 3T_m(k-1) + 2T_m(k-2) = 0$

The solution to this recurrence relation has the form

$T_m(k) = C_1 + C_2\, 2^k$ where C_1 and C_2 are some constants.

From $T_m(1) = \delta+\tau$; $T_m(2) = 3\delta+\tau$; $T_m(3) = 7\delta+\tau$; $T_m(4) = 15\delta+2\tau$, we can solve for C_1 and C_2, obtaining $C_1 = -\delta$ and $C_2 = \delta+ \tau/8$. Thus

$T_m(1) = \delta+\tau$; $T_r(1) = 2\delta$;

$T_m(2) = 3\delta+\tau$; $T_r(2) = 4\delta+\tau$;

For $k \geq 3$,

$T_m(k) = 2^k\delta + 2^{k-3}\tau - \delta$; $T_r(k) = 2^k\delta + 2^{k-3}\tau$.

Note that the time complexity of this algorithm is exponential in the rank of the lowest-ranked correct process.

3.2 Proofs of Properties

P1: Clear from the structure of the algorithm.

P2: It's clear from the structure of the algorithm that a process p starts to provide help only when p receives REQ from some process q, or p times out and becomes a helper by itself.

Case 1 If process p receives REQ from some process q, since REQ's are sent to processes in the increasing order by rank, q must have sent REQ's to all processes ranked lower than p and timed out waiting for help before sending this one. From the discussion in ***R2***, if some process p' ranked lower than p is correct, then q will receive DLV from p' before timing out waiting for help from p', so q would not send REQ to p, which is a contradiction.

Case2 If process p times out and becomes a helper by itself, then either p sent REQ to all lower-ranked processes and timed out waiting for help or p is ranked 1 and didn't send any REQ before starting to provide help. From the discussion in ***R1*** and ***R2***, in both cases all processes ranked lower than p must have crashed, otherwise they would have provided help before p timed out.

P3: It's clear from the structure of the algorithm that process p receives MSG only from the broadcaster or some helper q, where $q < p$.

Case 1 If process p receives MSG from the broadcaster, the broadcaster must have sent MSG to all processes ranked higher than p because MSG's are sent to processes in decreasing order by rank.

Case 2 If process p receives MSG from some helper q, where $q < p$, q either timed out and became a helper by itself or received REQ from some process $p' \geq p+1$ (the upper bound of the loop in line (3) implies q would not send MSG to p if $p' \leq p$). If q timed out and became a helper itself, it's clear from the structure of the algorithm that q would not send any MSG, a contradiction. If q received REQ from some process $p' \geq p+1$, q must have sent MSG to all processes ranked between p and p' because MSG's are sent to processes in decreasing order by rank. Since a process sends REQ only when it received MSG but timed out waiting for DLV, p' must have received MSG before sending the REQ. If p' received MSG from the broadcaster, MSG must have been sent to all processes ranked higher than p' as proved in *Case 1*. If p' received MSG from some helper that is different from q, following the same procedure as described above, we can prove (if $p' < N-1$) that process $p'' \geq p'+1$ also received MSG.

Thus, by induction, if a process p receives MSG, then MSG has been sent to all processed ranked higher than p.

P4: Each process sends REQ only to lower-ranked processes, so it's clear that if process p receives REQ from some process q, q is ranked higher than p. And if process p

didn't receive MSG from any process before receiving this REQ, from the discussion in ***R1*** and ***R2***, processes ranked between p and q either crashed or didn't receive MSG, otherwise they would have asked p for help and p would have sent DLV to q in time to prevent q from timing out and sending REQ to p.

P5: It's clear from the structure of the algorithm that a process p receives DLV only from lower-ranked processes, either the broadcaster or some helper.

Case 1 If process p receives DLV from the broadcaster b, then b sent MSG to all processes before sending that DLV. DLV's are sent in increasing order by rank, so the broadcaster sent DLV to all processes ranked lower than p, so all correct processes ranked lower than p have delivered or will deliver m.

Case 2 If process p receives DLV from some helper q, by **P2**, processes ranked lower than q have crashed. If q received MSG before executing the help procedure, by **P3**, MSG had been sent to all processes ranked higher q. If q didn't receive MSG, q must have received REQ from some process p'. Since a process sends REQ only when it received MSG but timed out waiting for DLV, p' received MSG before sending the REQ. By **P3**, MSG had been sent to all processes ranked higher than p', and it's clear from the structure of the algorithm that q would send MSG to all processes ranked between q and p' before sending any DLV. Thus MSG must have been sent to all correct processes. Note that helper q sends all messages in the same order as the broadcaster. DLV must have been sent to all correct processes ranked between q and p before being sent to p. By **P1,** all correct processes ranked between q and p, including q, eventually deliver the message. Thus, all correct processes ranked lower than p have delivered or will deliver the message.

4. Correctness of UTRB4

We now prove the correctness of algorithm UTRB4.

Theorem: ***Algorithm UTRB4 satisfies properties*** **B1-B4.**

Proof:

Validity: Assume that the broadcaster b is correct. It sends MSG at real time t_b to all other processes, and DLV by time $t_b+\tau$ to all other processes. Since the channels are FIFO, all correct processes other than b will first receive MSG and then receive DLV at most $\delta+\tau$ time units later, which is the smallest value of $T_m(p)$ for all $p \in PID\backslash\{b\}$. Thus, every correct process other than b delivers the message *m* without timing out while waiting for DLV from b. Clearly, b delivers *m*.

Integrity: Clear from the structure of the algorithm and the assumptions about the network.

Δ_b-Timeliness: Assume that totally *f* processes, including the broadcaster, crash during an execution of the protocol. From the discussion in section 3, we know that the time

when all correct processes deliver the message depends on the rank of the first correct process that received MSG or REQ. The worst case for the time complexity occurs when the broadcaster crashes immediately after sending MSG to the process ranked *N-1* and the processes ranked *1..f-1* crash before they can provide help. In the worst case, it takes δ time units for the MSG sent by the broadcaster to arrive at the process ranked *N-1,* and this process waits until it times out after sending REQ to each process ranked *1..f-1*, which takes time $T_m(N-1)+\Sigma_{j=1..f-1}T_r(N-1-j)$. Then it either delivers by itself (if N-f =1) or asks the process ranked *f* for help and delivers on receiving DLV from that process. When N-f >2, process ranked *f* needs to help processes ranked between *f* and *N-1* by sending MSG before sending any DLV.

Summarizing the above, we see that if *f* processes crash then any message broadcast at time *t* cannot be delivered after time $t+\Delta_b$, where:

If N-f = 1 then $\Delta_b = \delta + T_m(N-1)$

Else if N-f =2 then $\Delta_b = \delta + T_m(N-1) + \Sigma_{j=1..f-1}T_r(N-1-j)+2\delta$

Else $\Delta_b = \delta + T_m(N-1) + \Sigma_{j=1..f-1}T_r(N-1-j)+ 2\delta + \tau$

Uniform Agreement: This requirement is trivially satisfied if no process ever delivers a message. Suppose some process *d* delivers a message *m*. Note that *d* either delivered the message itself without receiving DLV, or received a DLV sent by another process *p*. By **Integrity**, *m* was broadcast by some process *b*. We need to show that all correct processes eventually deliver *m*. For contradiction, let *q* be a correct process that never delivers *m*.

Case 1: *d* delivered the message itself without receiving DLV. Then either *d* is *b* or *d* has received MSG or REQ before this delivery.

Case 1.1: *d* is *b.* i.e. *d* is the broadcaster. *d* must have sent MSG and DLV to all other processes before this self-delivery, so all correct processes receive MSG and DLV, thus eventually deliver *m*, including *q*, a contradiction.

Case 1.2: *d* received MSG before this delivery.

If rank (d, m.src) = rank (q, m.src), then d = q, so *q* delivered *m*, a contradiction.

If rank (d, m.src) < rank (q, m.src), by **P3**, MSG must have also been sent to *q*. Since *q* is correct, by **P1**, *q* eventually delivers *m*, a contradiction.

If rank (d, m.src) > rank (q, m.src), *d* must have sent REQ messages to all processes ranked lower than *d* before this self-delivery, so *q* must have received REQ from *d.* By **P1**, *q* eventually delivers *m*, a contradiction.

Case 1.3: *d* received REQ but didn't receive MSG before this delivery.

If rank (d, m.src) = rank (q, m.src), then d = q, so *q* delivered *m*, a contradiction.

If rank (d, m.src) ≺ rank (q, m.src). Since *d* sent DLV messages to all processes ranked higher than *d* before this self-delivery, *q* received DLV and thus delivered *m*, a contradiction.

If rank (d, m.src) ≻ rank (q, m.src), by **P2**, all processes ranked lower than *d* have crashed, including *q*, a contradiction.

Case 2: *d* received a DLV sent by another process *p*. It's clear from the structure of the algorithm that a process can only receive DLV from lower-ranked processes, so rank (p, m.src) ≺ rank (d, m.src).

Case 2.1: *p* is *b*. *p* must have sent MSG to all processes before sending DLV to *d*, so all correct processes receive MSG. By **P1**, all correct processes eventually deliver the message *m*, including *q*, a contradiction.

Case 2.2: *p* is a helper (i.e. p≠ b) and rank (q, m.src) ≺ rank (p, m.src) ≺ rank (d, m.src). By **P2**, *q* must have crashed, a contradiction.

Case 2.3: *p* is a helper and rank (p, m.src) ≤ rank (q, m.src) ≺ rank (d, m.src). Since *q* is correct, by **P5**, *q* delivers *m*, a contradiction.

Case 2.4: *p* is a helper and rank (p, m.src) ≺ rank (d, m.src) ≤ rank (q, m.src). Before providing help, *p* either timed out without receiving REQ from any process or received a REQ sent by some process *r*.

Case 2.4.1: *p* timed out without receiving REQ from any process.
In this case, *p* must have received MSG before providing help. By **P3**, MSG must have been sent to all processes ranked higher than *p*, including *q*. Since *q* is correct, by **P1**, *q* eventually delivers *m*, a contradiction.

Case 2.4.2: *p* received REQ from *r* and rank (r, m.src) ≤ rank (q, m.src).
r must have received MSG before sending the REQ message. By **P3**, MSG was sent to all processes ranked higher than *r*, including *q*. Since *q* is correct, by **P1**, *q* eventually delivers *m*, a contradiction.

Case 2.4.3: *p* received REQ from *r* and rank(r, m.src) ≻ rank (q, m.src).
By ***R1*** and ***R2*****,** *p* didn't receive MSG, otherwise *p* would time out earlier than *r*, as discussed in *Case 2.4.1*. By **P4**, processes ranked between *p* and *r* either crashed or didn't receive MSG. So before sending DLV to any process, *p* must have sent MSG to all processes ranked between *p* and *r*, including *q*, so *q* received MSG. By **P1**, *q* eventually delivers *m*, a contradiction.

5. Message Complexity of UTRB4

Assume that including the broadcaster (ranked 0), totally *f* processes crash during an execution of the protocol. The following properties are useful in the analysis of the message complexity of UTRB4.

MP1 A process can only receive MSG and DLV from lower-ranked processes and REQ from higher-ranked ones.
MP2 A process sends REQ messages only if it did not receive DLV after receiving MSG, and only to those processes ranked between itself and the sender of the MSG it received.
MP3 DLV is sent to each process at most once.
MP4 Each correct process receives MSG and REQ each at most once.

Proofs:
MP1, **MP2:** Clear from the structure of the algorithm.

MP3: Proof by contradiction. Suppose two DLV messages are sent to some process *p*, then the second DLV must be sent by some helper *q* because both the broadcaster and helpers send DLV at most once to each other process. By **MP1**, *q* is ranked lower than *p*. Clearly *q* was alive when *p* received the first DLV. Let *s* be the process that sent the first DLV to *p*; as argued above, $s \neq q$.

Case 1 If *s* is ranked higher than *q*, then *s* is a helper. By **P2,** *q* had crashed by the time *s* sent DLV to *p*, which is a contradiction.

Case 2 If *s* is ranked lower than *q* and *s* sent DLV to *q* before sending DLV to *p* (because DLV can only be sent in increasing order by rank), then *q* would start to provide help only if *q* received REQ from some correct process ranked higher than *p*. It's clear from the structure of the algorithm that *q* doesn't send DLV to process *p* in this case. That's a contradiction.

Case 3 If *s* is ranked lower than *q* and *s* didn't send DLV to *q* before sending DLV to *p*, then *s* is a helper and clearly *s* received MSG before receiving REQ from some process *r* that is ranked higher than *q*. Since MSG's are sent to processes in decreasing order by rank, *q* also received MSG. Since *q* was still alive at that time, from the discussion in ***R1*** and ***R2***, we know that *q* would time out and send REQ to *s* earlier than *r*. That's a contradiction.

MP4: On receiving REQ, a correct process helps all correct processes deliver the message. By ***R1*** and ***R2***, a helper has enough time to finish the help procedure before the next REQ is sent. By **MP2**, this helper receives REQ at most once.

Both the broadcaster and helpers send MSG's to other processes at most once. A helper *q* sends MSG only when it received REQ from some process *p* but didn't receive MSG before, and to processes ranked between *q* and *p*. By **P4**, all correct processes in this range didn't receive MSG before. So a correct process receives MSG at most once.

5.1 Computation of Message Complexity

Worst Case: Note that in the case of no failures, totally *2(N-1)* messages are sent per broadcast. By **MP1** and **MP2,** for a process ranked i, the numbers of REQ's it sends and

extra MSG's that are sent to it sum to at most i-1. By **MP3**, at most *N-1* DLV's are sent in this system.

In the worst case, the *f* lower-ranked processes crash one by one in increasing order by rank and just at the time that causes the maximum number of extra messages. Thus the process of rank *f* (note that the broadcaster has rank 0) is the first correct process that takes the role of the initial broadcaster. It sends *f-1* REQ messages before becoming a helper by itself.

Totally, the maximum number of messages is:

$$2(N-1)+1+2+...+(f-1) = 2(N-1)+ f(f-1)/2;$$

Best Case:

Case f < N: The best case occurs when the process of ranked 0 or 1 is correct. It times out first after receiving MSG from the initial broadcaster and becomes a helper immediately. In this case, total number of messages that are sent in the system is the same as the case of no failures, that is: *2(N-1).*

Case f = N: The best case occurs when all processes fail at the start of execution, in which case the number of messages sent is 0.

5.2 Message Complexity assuming Correlated Failures

We would like to argue that in practice, the situation as described in the worst case seldom happens. If multiple processes crash, they are likely to crash at about the same time. We analyze the message complexity under this assumption.

Worst Case:

The process of rank *f* is the first correct process that takes the role of the initial broadcaster, and all *f* lower-ranked processes crashed immediately after the broadcaster sent DLV to the process of rank *f-1.* Maximum number of messages here is: *2(N-1)+ f-1*.

Best Case:

Case f < N: The best case occurs when the process of rank 0 or 1 is correct. Total number of messages that are sent here is the same as the case of no failures, that is: *2(N-1).*

Case f = N: The best case occurs when all processes fail at the start of execution, in which case the number of messages sent is 0.

Average Case:

We can see that under this restriction, the number of extra messages only depends on the number of REQ messages sent by the first correct process that eventually takes the role of the initial broadcaster. If this first correct process has rank *x*, then the total number of extra messages is *x-1*. Consider all possible combinations of the *f* crashed processes, we get the average number of messages that are sent is: $2N+\Sigma(x-1)\binom{n-x-1}{f-x}/\binom{n}{f}$, where x ranges from 1 to f.

BIBLIOGRAPHY

[Babaoglu and Toueg] Ozalp Babaoglu and Sam Toueg, Non-Blocking Atomic Commitment. Chapter 6 of Sape Mullender, editor, *Distributed Systems*, 2nd edition. Addison-Wesley, 1993. An extended version appeared as: Ozalp Babaoglu and Sam Toueg, Understanding Non-Blocking Atomic Commitment. University of Bologna, Laboratory for Computer Science, Technical Report UBLCS-93-2, 1993.