# Software-Failure-Tolerant and Highly-Available Design of the Distributed Emergency Response Management System

November 11, 2024

| Name | Student ID | Module |
|---|---|---|
| Shaza Ahmed | TODO | TODO |
| Sam Anthony | 40271987 | Sequencer |
| Bence Matajsz | TODO | Testing |
| Vijaykumar Patel | TODO | Front end |

## 1 Introduction

TODO

## 2 Architecture

TODO

## 3 Front end

TODO

## 4 Replica manager

TODO

## 5 Sequencer

The purpose of the sequencer is to ensure that requests are delivered to all of the replicas in the same (total) order. This ensures sequential consistency of the system because all replicas perform the same computations in the same order.

The FE delivers requests to the RMs via *totally ordered reliable multicast*. When the FE receives a request from a client, it sends the request to the multicast group. All of the RMs, as well as the sequencer, are members of the group, so they receive the request.

When a RM receives a request, it does not deliver it to the replica immediately. Rather, it adds the message to a *holdback queue* and waits to receive an *order message* from the sequencer.

The sequencer is also a member of the multicast group, but it has a special function. When it receives a message, it multicasts an *order message* containing the ID of the message it has just received.

When a RM receives an order message, it inspects the ID attached to it and removes the associated message from the holdback queue and delivers it to the replica.

The FE and RMs need not be aware of any of this behavior because it is encapsulated in the totally-ordered reliable multicast class. As far as the FE knows, it is just “sending a request to all of the RMs.” And from the perspective of an RM, it is just “receiving a request from the FE.” They can be sure that they are reliably sending and receiving requests in the proper order, without worrying about the implementation details, or even knowing of the existence of the sequencer.

The totally ordered multicast protocol is built on top of a reliable multicast mechanism—an implementation of the Trans Protocol, “[which] uses a combination of positive and negative acknowledgement strategies to achieve efficient reliable broadcast or multicast communication” [1].

**Sequencer pseudocode:**

```
type dataMsg struct {
  id int
  data
}

type orderMsg struct {
  id int
  seq int
}

class groupMember {
  seq := 0
  holdback := new Queue[dataMsg]
  deliver := new Queue[dataMsg]
  rMulticast := new Trans

  multicast(m dataMsg) {
    rMulticast.send(m)
  }

  on rMulticast.recv(m dataMsg) {
    holdback.enqueue(m)
  }

  on rMulticast.recv(m orderMsg) {
    wait until m.id in holdback && seq == m.seq
    data := delete(m.id, holdback)
    deliver.enqueue(data)
    seq++
  }
}

// Sequencer is also a member of the multicast group.
class sequencer {
  seq := 0
  rMulticast := new Trans

  on rMulticast.recv(m dataMsg) {
    order := orderMsg{m.id, seq}
    rMulticast.send(order)
    seq++
  }
}
```

**Reliable multicast (Trans) pseudocode:**

```
// message
type m struct {
  id mid
  sender pid
  seq int // sequence number
  positiveAcks []mid
  negativeAcks []mid
  data
}

type mid ”message-ID”
type pid ”process-ID”

var (
  positiveAcks []mid
  negativeAcks []mid
  received []m
  retransmissions []mid
  lastSend time
)

send(m) {
  pkt := (m, positiveAcks, negativeAcks)
  multicast(pkt)
  positiveAcks = []
  go timeout(m)
  lastSend = now()
}

timeout(m) {
  sleep until timeout

  if m not in positiveAcks {
    insert(m, retransmissions)
  }
}

recv(m) {
  insert(m.id, positiveAcks)
  insert(m, received)

  if m.id in negativeAcks {
    delete(m.id, negativeAcks)
  }
  if m.id in retransmissions {
    delete(m.id, retransmissions
  }

  for each mid in m.positiveAcks {
    delete(mid, positiveAcks)
    if mid not in received {
      insert(mid, negativeAcks)
    }
  }

  for each mid in m.negativeAcks {
    if mid in received {
      insert(mid, retransmissions)
    } else {
      insert(mid, negativeAcks)
    }
  }

  acks := union(positiveAcks, negativeAcks)
  for each ack in acks s.t.
      sender(ack) == sender(m)
      && m.seq > ack.seq+1 {
    insert(m.id, negativeAcks)
  }
}

retransmit() {
  forever {
    wait until
      timeSince(lastSend) > threshold
      && len(retransmissions) > 0

    mid := pop(retransmissions)
    m := received[mid]
    send(m)
  }
}

// Observable Predicate for Delivery.
// The process that broadcast c has received and acked
// message a at the time of broadcasting c.
// All assertions must hold in order to return true.
OPD(a, c m) bool {
  assert (t.e. sequence [a, ..., c])
  for each i, m in sequence, except a {
    predecessor := sequence[i-1]
    assert (
      predecessor in m.positiveAcks
      || m.sender == precessor.sender)
    assert (m not in c.negativeAcks)
  }
}

// Partial order.
// All assertions must hold in order to return true.
(c m) follows(b m) bool {
  assert OPD(b, c)
  for all a in received {
    if OPD(a, b) {
      assert OPD(a, c)
    }
  }
}
```

## 6 Testing

TODO

## References

[1] P.M. Melliar-Smith, L.E. Moser, and V. Agrawala. "Broadcast protocols for distributed systems". In: *IEEE Transactions on Parallel and Distributed Systems* 1.1 (1990), pp. 17–25. DOI: `10.1109/71.80121`.