Classical Distributed Algorithms with DDS

Sara Tucci-Piergiovanni, PhD
Researcher
CEA LIST

Angelo Corsaro, PhD
Chief Technology Officer
PrismTech
Outline

- DDS and QoS, properties of streams and local caches
- Advanced properties on local caches: the eventual queue
  - Implementation of the eventual queue based on Lamport’s distributed mutual exclusion algorithm
  - Dealing with failures, mutual exclusion implemented as a Paxos-like algorithm
- Concluding Remarks
DDS and QoS, properties of streams and local caches
• DDS let multiple writers/readers produce and consume streams of data, like this:
QoS: properties on streams (1/3)

- legal stream of reads **Reliability Policy = Best Effort**

DataWriter

\[
\text{w (1)} \quad \text{w (2)} \quad \text{w (3)}
\]

DataReader

\[
\text{r (): 1} \quad \text{r (): nil} \quad \text{r (): nil}
\]

Proactive read, only new values
Non-blocking write
QoS properties on streams (2/3)

- Legal stream of reads if **Reliable Policy = Reliable** → *the last value is eventually read*
• legal stream of reads if **Reliable Policy = Reliable, History = keepAll** → *all the values are eventually read*

```
dataWriter
w (1) w (2) w (3)
```

```
dataReader
r () : 1 r () : nil r () : [2, 3]
```

**History defines how many ‘samples’ to keep in the local cache**
Local Caches and Message Arrivals

The update arrives after the read.
Local Caches and Message Arrivals

- **Writer crash** - eventual semantics is not guaranteed: *dReader2 misses 3*

The update arrives after the read.
Fault-Tolerant Reliable Multicast

• First useful abstraction to guarantee eventual consistency in case of writer failure

• Many possible implementations ranging from deterministic flooding to epidemic diffusion.
  - History = KeepLast(1)
    - Possible Implementation: Push with Failure Detectors: each process (data reader) relays the last message when it suspects the writer to be failed (optimizations are possible).
  - History = keep all
    - Sending the last value does not suffice, local caches should be synchronized from time to time

• Let us remark that different protocols could be implemented. Depending on the history QoS setting the best suited protocol will be employed.

• However, FT Reliable Multicast is best implemented as an extension of the DDSI/RTPS wire protocol. Some DDS implementations, such as OpenSplice, provide FT Reliable Multicast as yet another level of Reliability Policy

• In the context of this presentation we focused on user-level mechanisms
Local Caches benefit of eventual consistency in absence of failures

DDS provides an eventual consistency model where $W=0$ (number of acks expected before completing a write) and $R=1$ (number of « replicas » accessed to retrieve data). This means that data is eventually written on all « destinations » but is only read locally.

With failures, eventual consistency only implementing a fault-tolerant reliable multicast

What about stronger properties on caches? Let’s try to implement a global queue

- producers
- consumers
Advanced properties on local caches: the eventual queue
Properties on caches

Local Caches benefits of eventual consistency in absence of failures

With failures, eventual consistency only implementing a fault-tolerant reliable multicast

What about stronger properties on caches? Let’s try to implement a global queue
Local Caches benefits of eventual consistency in absence of failures

With failures, eventual consistency only implementing a fault-tolerant reliable multicast

What about stronger properties on caches? Let’s try to implement a global queue
Local Caches benefits of eventual consistency in absence of failures

With failures, eventual consistency only implementing a fault-tolerant reliable multicast

What about stronger properties on caches? Let’s try to implement a global queue
Properties on caches

Local Caches benefits of eventual consistency in absence of failures

With failures, eventual consistency only implementing a fault-tolerant reliable multicast

What about stronger properties on caches? Let’s try to implement a global queue
Properties on caches

Local Caches benefits of eventual consistency in absence of failures

With failures, eventual consistency only implementing a fault-tolerant reliable multicast

What about stronger properties on caches? Let’s try to implement a global queue
Properties on caches

Local Caches benefits of eventual consistency in absence of failures

With failures, eventual consistency only implementing a fault-tolerant reliable multicast

What about stronger properties on caches? Let’s try to implement a global queue
Eventual Consistent Queue

We are not interested in guaranteeing one-copy serializability:

- If a process performs \texttt{enq}(a) at some point \( t \) and the queue is empty, the subsequent \texttt{deq}() will get \( a \).
- If a process performs \texttt{deq}(a), no other process will perform \texttt{deq}(a)

Serializability would seriously limit concurrency

We propose a weaker, but still useful, semantics for the queue: Eventual Queue

- (**Eventual Dequeue**) if a process performs an \texttt{enq}(a), and there exists an infinite number of subsequent \texttt{deq}(), eventually some process will perform \texttt{deq}(a).

- (**Unique Dequeue**) If a correct* process performs \texttt{deq}(a), no other process will perform \texttt{deq}(a)

*correct process=process that never crashes
(Eventual Dequeue) if a process performs an \( \text{enq}(a) \), and there exists an infinite numbers of subsequent \( \text{deq}() \), eventually some process will perform \( \text{deq}(a) \).
(Eventual Dequeue) *if a process performs an* \( \text{enq}(a) \), *and there exists an infinite numbers of subsequent* \( \text{deq}() \), *eventually some process will perform* \( \text{deq}(a) \).

The order in which values are de-queued is not guaranteed to be the order in which they have been enqueued. Some value \( \bullet \) enqueued after \( \bigcirc \) could be de-queued before \( \bullet \), but eventually each value will be de-queued.
Implementing the Eventual Queue with DDS

At implementation level the eventual queue is implemented through local caches.

Abstraction level

eventual queue

.dataReader1

.dataReader2

.dataReader3

Implementation level, DDS primitives available

write() Local cache for dataReader1

write() Local cache for dataReader2

write() Local cache for dataReader3

If the pink circle is consumed by the application, it must be removed from other caches before a new dequeue will be performed.

.data Readers on topic T

data Writers on topic T

Distributed mutual exclusion is needed to consistently consume samples!
• In terms of programming API our distributed Queue implementation is specified as follows:

```scala
abstract class Queue[T] {
  def enqueue(t: T)
  def dequeue(): Option[T]
  def sdequeue(): Option[T]
  def length: Int
  def isEmpty: Boolean = length == 0
}
```

• The operation above have exactly the same semantics of the eventual queue formally specified a few slides back
Implementing the Distributed Queue in DDS
A Distributed Mutual Exclusion Based Distributed-Queue

- Different distributed algorithms can be used to implement the specification of our **Eventual Queue**
- In a first instance we’ll investigate an extension of Lamport’s Distributed Mutual Exclusion for implementing an Eventual Distributed Queue
- In these case the **enqueue** and the **dequeue** operations are implemented by the following protocol:

**enqueue():**
- **Do a DDS write**

**dequeue():**
- If the “local” cache is empty then return “None”
- Otherwise start the Distributed Mutual Exclusion Algorithm
- Once entered on the critical section, pick an item from the queue
- Ask all other group members to POP this element from their “local” cache
- Exit the critical section and ACK other member if necessary
- Return the data to the application
A Distributed Mutual Exclusion Based Distributed-Queue

- Data readers will issue a request to perform a take on their own local cache (set of requesters)
- The same set of data readers will acknowledge the access to the local cache (set of acceptors)

Assumptions
- We need to know the group of requesters/acceptors in advance, a total order on their IDs must be defined
- FIFO channels between data readers
- No synchronization between clocks, no assumptions on bounds for message delays: the algorithm is based on *logical clocks*
Implementation 1 - A possible run

**app 1**
- **deq() : a**
  - req{ts, (1,1)}
  - 1
  - (1,1)
  - (1,2)
- pop{ts, (1,3)}
  - ack{ts, (4,2)}
  - 4
  - (3,2)

**app 2**
- **deq() : b**
  - req{ts, (1,2)}
  - 1
  - (1,2)
  - (1,1)
  - (1,2)
- pop{ts, (1,3)}
  - req{ts', (3,2)}
  - 3
  - ack{ts, (2,2)}

**Note:** The diagram illustrates a possible run of the implementation with sequences of actions and acknowledgments between applications and their requests and responses.
In DDS Terms..

- To implement this algorithm it is required that
  - DEQUEUE/ACK/POP messages are FIFO
  - POPs issued by a member are received before its ACKs that release control
- This leads to use a single topic/topic type for all commands and a single data-writer for writing them
- The queue implementation uses only two topics defined as follows:
  - Topic\(\text{name} = \text{QueueElement}, \text{type} = \text{TQueueElement}, \text{QoS} = \{\text{Reliability.Reliable, History.KeepAll}\}\)
  - Topic\(\text{name} = \text{QueueCommand}, \text{type} = \text{TQueueCommand}, \text{QoS} = \{\text{Reliability.Reliable, History.KeepAll}\}\)

```c
1 typedef sequence<octet> TData;
2 struct TQueueElement {
3     TLogicalClock ts;
4     TData data;
5 };
6 #pragma keylist TQueueElement

9 enum TCommandKind {
10     DEQUEUE,
11     ACK,
12     POP
13 };
14
15 struct TQueueCommand {
16     TCommandKind kind;
17     long mid;
18     TLogicalClock ts;
19 };
20 #pragma keylist TQueueCommand
```
Let’s see what it would take to create a multi-writer multi-reader distributed queue where writers enqueue messages that should be consumed by one and only one reader.
• (Any) Message Producer

```scala
def main() {
  val group = Group(gid)
  group.join(mid)
  println("Producer:> Waiting for stable Group View")
  group.waitForViewSize(n)

  val queue = Queue[String](mid, gid, n)

  for (i <- 1 to samples) {
    val msg = "MSG["+ mid +", "+ i +"]"
    println(msg)
    queue.enqueue(msg)
    // Pace the write so that you can see what's going on
    Thread.sleep(300)
  }
}
```
• (Any) Message Consumer

```scala
val group = Group(gid)
group.join(mid)
println("Producer:> Waiting for stable Group View")
group.waitForViewSize(n)

val queue = Queue[String](mid, gid, n)

while (true) {
    queue.sdequeue() match {
        case Some(s) => println(s)
        case _ =>
    }
}
```
def deq() =
    lclk = (0, mid)
    currentRequestLClk = (inf, mid)
    send (DEQUE, ++lclk) to all readers

    wait_acks(n)
    take() the sample with min (ns, wid)
    send (POP, (sn, wid)) to all readers
    send (ACK, ++lclk) to all readers in requestQueue

def onDequerequest =
    lclk = max(llclks_received, lclk))
    if (currentRequestClk >
        logicalClock_received(i)
        send an ack
    else
        add request to requestQueue

def onPopRequest =
    When receive a pop execute the request (pop).
Dealing with Failures… Properly!
• What about the algorithm if a process can crash?
• Let us consider the possible ‘blocking points’ if processes can crash, where N is the number of processes

POINT 1: granting the access in ME
To make progress we need an acknowledgement from everyone. Only one failure out of N during requesting will block the protocol

POINT 2: releasing the ‘lock’
To make progress we need the process p3 to eventually send the pop message. A crash of P3 will block the protocol

How to solve these points?
Towards Implementation 2 - Failures

- What about the algorithm if a process can crash?
- Let us consider the possible ‘blocking points’ if processes can crash, where N is the number of processes

**POINT 1: granting the access in ME**
To make progress we need an acknowledgement from everyone. Only one failure out of N during requesting will block the protocol

**POINT 2: releasing the ‘lock’**
To make progress we need the process p3 to eventually send the pop message. A crash of P3 will block the protocol

**SOLUTION**
1) The access is granted by a quorum, i.e. a majority
2) Assumption on failures: at most f with N=2f+1.
3) One leader that is in charge to serialize the requests. Concurrent request must be (eventually) avoided to eventually get a quorum
Towards Implementation 2 - Failures

- What about the algorithm if a process can crash?
- Let us consider the possible ‘blocking points’ if processes can crash, where N is the number of processes

**POINT 1: granting the access in ME**
To make progress we need an acknowledgement from everyone. Only one failure out of N during requesting will block the protocol.

**POINT 2: releasing the ‘lock’**
To make progress we need the process p3 to eventually send the pop message. A crash of P3 will block the protocol.

**SOLUTION**
1) The leader will multicast the pop message when it receives it from the requester, otherwise it will kill the requestor.
2) The underlying protocol electing the leader assures that eventually a correct process will be the leader (multiple leaders are possible) then the multicast will have success.
Implementation 2: Paxos-like algorithm

The Paxos algorithm lies on the following assumptions:

- We need to know the group of acceptors in advance (not necessarily the group of requesters).
- In the set of acceptors there exists a majority of correct processes.
- Leaders are chosen among a set of ‘proposers’, we need to know the group of proposers in advance.
- Each process is equipped with an oracle $\Omega$ (eventual leader), which eventually output the same correct proposer as leader at each process (very simple to implement! each process choose the proposer with the minimum id from the list of proposers not suspected to have crashed).
Implementation 2: Requester pseudo-code

init()
  pending=nil;
  read local queue and take the minimum sequence number sn for a sample s from a writer k (round robin fashion)
  send (request('deq', ts=(sn,k,my_id)), to currentLeader=Ω.leader());
  pending=request('deq', ts=(sn,k,my_id));

|| when Ω.leader() != currentLeader and pending!=nil
  send pending currentLeader=Ω.leader();

|| when receive(ack('deq', ts_rcv)) from currentLeader
  If pending contains a request s.t. ts==ts_rcv
    take(s,ts) from the local queue
    send (notify ('take_done', ts) to currentLeader
    pending= notify ('take_done', ts)

|| when receive (notify('take_done', ts_rcv)) from currentLeader
  if notify('take_done', ts_rcv) ∈ pending then pending=nil
  else take(s,ts_rcv) from the local queue
  if pending contains a request s.t. ts==ts_rcv then restart the protocol.
Implementation 2 - Proposer pseudo-code

```plaintext
Init()
   pending;
   grants;

|| when receive (request('deq', ts=(sn,k,my_id))
   if pending!=nil;
      while grants < n+1/2 {
         round++
         send (request('deq', ts=(sn,k,requester_id)), round) to acceptors
         pending=ts
         wait for N+1/2 reply (ack_rcv, ts_rcv, round) }
      send (ack('deq', ts) to requester_id

|| when receive (reply(ack_rcv, ts_rcv), round) from p_j
   if pending! nil
      if (ts= =ts_rcv ) then grants++

|| when receive (notify 'pop', ts_rcv)
   send (notify 'pop', ts_rcv) to all requesters
   if pending contains a request s.t. ts==ts-rcv, then remove the request from pending
```
Concluding Remarks

• DDS provides an eventual consistency model where $W=0$ (number of acks expected before completing a write) and $R=1$ (number of « replicas » accessed to retrieve data). This means that data is eventually written on all « destinations » but is only read locally – assuming no crashes.

• DDS does not provide fault-tolerant multicast, meaning that under writer fault the reader can remain eventually inconsistent.

• Starting from this weak semantics, higher level primitives can be built very effectively to facilitate the development of distributed applications that require complex coordination mechanisms such as:
  - Multi-Reader / Multi-Writer Distributed Eventual Queue
  - Mutual Exclusion
  - Eventual Leader Election

• Our experience with the DADA toolkit is that the combination of DDS and Scala made our algorithm performant and very elegant and compact.

• Finally, the DADA toolkit provides useful primitives with well specified semantics that can be used by to greatly ease the creation of sound fault-tolerant distributed systems.
All the algorithms presented were implemented using DDS and Scala.

Specifically, we’ve used the OpenSplice Escalier language mapping for Scala.

The resulting library has been baptized “DADA” (DDS-based Advanced Distributed Algorithms) and is available under LGPL-v3.
Thank you for your attention
Questions?