Lecture 23

In Lecture 22 we studied simple handshake protocols and leader election in a ring. We used *event based specifications*. In this lecture we examine consensus protocols, mainly simple consensus, also called 2/3 consensus for reasons that will become clear as we go.

Consensus is a critical protocol for distributed databases and file systems. It is at the heart of cloud computing. The MIT lecture in the material from Lecture 22 is also appropriate here because it presents the 2/3 consensus protocol. There is also a publication about proving that this protocol is correct using Nuprl that is included as supplemental material: Formal Specification, Verification, and Implementation of Fault-Tolerant Systems. This paper is somewhat advanced, but parts of it are accessible at this point, e.g. Section 2. Note, we now prefer the term *event handler* rather than event class. This article is evidence that constructive type theory is a good formalism for specifying protocols and proving them correct.

Consensus protocols are important components of distributed services based on replicas. Consensus is used to achieve agreement among independent replicas. Here is a typical diagram used to explain their role.
Fault-Tolerant Distributed Systems - a scenario
Assume at most $f$ failures

For simple 2/3 consensus we need $3 \cdot f + 1$ replicas

The replicas might receive the broadcast messages in different orders. The message will be a slot number and a command as broadcast by Aneris as it receives requests from clients and proposes them in order.

<table>
<thead>
<tr>
<th>proposals</th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1, c_1)$</td>
<td>$(2, c_2)$</td>
<td>$(3, c_3)$</td>
<td>$(2, c_2)$</td>
<td></td>
</tr>
<tr>
<td>$(2, c_2)$</td>
<td>$(3, c_3)$</td>
<td>$(1, c_1)$</td>
<td>$(1, c_1)$</td>
<td></td>
</tr>
<tr>
<td>$(3, c_3)$</td>
<td>$(1, c_1)$</td>
<td>$(2, c_2)$</td>
<td>$(4, c_4)$</td>
<td></td>
</tr>
<tr>
<td>$(4, c_4)$</td>
<td>$(4, c_4)$</td>
<td>$(4, c_4)$</td>
<td>$(3, c_3)$</td>
<td></td>
</tr>
</tbody>
</table>

The replicas need to agree on an ordering of the commands so that there is a single serializable record of updates to the database or file system. They do this by voting using a single quorum
of $2 \times f + 1$ votes, e.g. by tolerating $f$ failures. They vote to decide on an ordering of the commands and when a command is decided on, they all execute it. The voting pattern is:

- propose the $n^{th}$ command
- see if a quorum agrees
  - if yes, vote for that command
  - if no, then vote again in another round, more data might have arrived at replicas
- when a command is decided notify all replicas and the client

The different kinds of internal messages have unique headers, e.g. “propose” and “notify” and “vote”.

We will write a 2/3 consensus in more detail next and consider how to specify it using concepts from our constructive type theory of events.

Properties of protocols for asynchronous distributed systems are usually classified as safety and liveness properties. Fred Schneider showed the value of this classification. Safety properties tell us that “nothing bad” will happen while liveness says that “something good” happens. Here are some properties we want. (See my MIT lecture (posted from Lecture 22) for details.)

P1. If inputs are unanimous, with value $v$, then any decision must have value $v$.
$$\forall v : T. (\forall e : E(Input).Input(e) = v \Rightarrow \forall e : E(Decide).Decide(e) = v)$$

P2. All decided values are input values.
$$\forall e : E(Decide). \exists e' : E(Input). (e' < e & Decide(e) = Input(e'))$$

P3. (Agreement - a safety protocol)
Any two decisions have the same value.
$$\forall e_1, e_2 : E(Decide).Decide(e_1) = Decide(e_2)$$

**Approaches to an algorithm**

Have each $P_i$ vote, send to all other $P_j$. Each $P_j$ collects a list of $n$ votes (for $P_1, \ldots, P_n$ processes). Use any deterministic choice function on the list $L$ of $n$ votes (say even ordered by process number). Let $f(L)$ be the consensus value.

But processes can fail! Say up to $f$ of them, and we don’t know which ones. What about collecting $2 \times f + 1$ votes only? We don’t know which $2 \times f + 1$?!

We now consider “pseudo code” for the 2/3 consensus approach, which is to collect a quorum of $2 \times f + 1$ votes and decide by *majority vote*. This quorum approach assures that the uncounted $f$ votes cannot overturn the majority if it is unanimous. Here is pseudo code for this approach - see the MIT lecture.
Ambitious students can read the actual Nuprl EventML code which has been formally verified and used in working systems. The article we published in the IEEE WRPE conference on rigorous protocol engineering is posted with this lecture.

Begin $r \in \mathbb{N}$, decided$_i$, vote$_i$:$\mathbb{B}$
$r := 0$; decided$_i := \text{false}$
% $v_i$ input to $P_i$, vote$_i = v_i$

Until decided$_i$ do:

1. $r := r + 1$
2. Broadcast $\text{vote}(r, vote_i)$ to group $G$
3. Collect $2 * f_1$ votes on list $L$
4. $vote_i := \text{majority}(L)$
5. If $\text{unanimous}(L)$ then decided$_i := \text{true}$

End

Elements of EventML protocol for 2/3 consensus (see article from WRPE conference)

Main SC

\[
\text{event\_handler} \quad \text{Replica} = \text{NewVoters} \gg= \text{Voter}
\]

\[
\text{Rounds}(n, c) = \text{Round}((n, c), c) \mid \\
\quad (\text{NewRounds} n \gg \text{Round});
\]

\[
\text{Voter}(n, c) = \\
\text{Rounds}(n, c) \text{ until Notify } n \mid \\
\quad (\text{Notify } n)
\]

\[
\text{Round}((n, r), c) = \\
\text{Output}((\text{loc}.'\text{vote}'\text{broadcast} \text{ reps} (((n, r), c), \text{bool}) \mid \\
\quad \text{Once}(\text{Quorum}(n, r))
\]

\[
\text{QuorumSide}(n, r) = \\
\text{Memory}((\text{loc}.[], []).\text{add}\_to\_qurorum}(n, r), \\
\quad \text{vote'base});
\]