Introduction

When we look at a distributed system we want each processor in the system to know what is the state of all the other processors and communication channels of the system. However, each processor can only tell what is its own state, and the information about the other processors must be somehow collected and form a meaningful and informative picture of the whole system. This “big picture” we are putting together from the different system components is called a snapshot.

Motivation

Why do we need a snapshot, and what exactly is a meaningful and informative snapshot? To answer that lets define the Stability property of a system.

System stability:

Let $f$ be a predicate function defined on the global states of a distributed system $D$. that is, $f(S)$ is true or false for a global state $S$ of $D$. We will say that $f(S)$ is a stable property of $D$ if $f(S)$ implies $f(S')$ for all global states $S'$ of $D$ reachable from global state $S$ of $D$. The system $D$ is stable if $f$ is true in one of it’s states since if $f$ is true at a point in a computation of $D$, then $f$ is true at all later points in that computation. Examples of stable properties are “computation has terminated”, “the system is deadlocked” and “all tokens in a token ring have disappeared”.

The snapshot of the system is then needed for us to determine the stability of the system, and it will be an informative snapshot if we can tell from it that the system is stable.

The distributed model

The system model we will work with is as follows:

- The complete system will be represented by a directed Graph, the vertices will represent the processors and the edges between them the communication channels and their direction.
- The processors in the system can perform two kinds of operations one is changing the processor state and the other is receiving/sending a message from/to a communication channel.

The system will work under the following assumptions:

- There is no synchronization between the processors (there are no clocks in the system).
- The communication channels have an infinite buffer for messages in the channel.
- The communication channels are error free - messages are not lost or changed in any way while in the channel.
- The communication channels deliver messages in the order they were sent (a FIFO buffer).
- A message in a communication channel can be delayed for an arbitrary but finite time or in other words, all messages sent in the channel will eventually arrive at their destination.
Formal definitions of the system

- **State of a communication channel** - the state of the channel is the sequence of messages contained in that channel buffer. That is, all the messages sent in the channel excluding the messages received from the channel.

- **State of a processor** - a single element from some finite set. Processors can only be in one state at a time and each processor in the system may have (but not obliged to have) a different set for it’s states.

- **Event** - an event is an atomic action of one processor in the system. During the event a processor may change its state and it may send/receive a message throw one communication channel at most that is directed away/to the processor. We will denote an event \( e \) in the system with the tuple: \( < p, s, s', M, c > \) where:
  - \( p \) - the processor in which the event occurred.
  - \( s \) - the state of \( p \) before the event.
  - \( s' \) - the state of \( p \) after the event (\( s' \) may be equal to \( s \)).
  - \( c \) - the channel whose state was changed by the event (may be null to indicate no channel was involved in the event).
  - \( M \) - the message sent/received from/to \( p \) throw the channel \( c \) (may be null to indicate no message was involved in the event).

- **Global state** – The global state of the system is the set of all the processors current states and the channels current states. We will also define the initial global state – a global state where each processor is in its initial state and each communication channel is in an empty state.

- **Next** - We will define how the system moves between states by defining the function \( \text{Next} () \). The function \( \text{Next} (S, e) \) value will be the global state immediately after the occurrence of the event \( e \) in the global state \( S \). We notice that \( \text{Next} () \) is defined only if event \( e \) can occur in the global state \( S \). In other words, for a global state \( S \), and an event \( e =< p, s, s', M, c > \) if \( \text{Next} (S, e) = S' \) then the state of \( p \) in \( S' \) is \( s' \) the state of the channel \( c \) in \( S' \) is its state in \( S \) with the message \( M \) added to its tail or removed from its head (depending if it was a receiving or sending of a message).

- **Computation of the system** – A sequence of events in the system that can occur one after the other. Formally, given a sequence of events \( \text{seq} = (e_0, e_1, \ldots, e_i, \ldots, e_m) \) \( \text{seq} \) is a computation of the system iff event \( e_i \) can occur in state \( S_i \) and \( \text{Next} (S_i, e_i) = S_{i+1} \) (\( S_0 \) is the initial global state).

Definitions example

We will demonstrate the system definitions with an example. The demonstrated system topology is of two processors \( p, q \) the processors are connected to each other with two communication channels: \( c : (p \rightarrow q), c' : (q \rightarrow p) \) see figure 1.

![Figure 1: The single token conservation system topology](image)

Each processor has two possible states:

- \( s_0 \) - no token
- \( s_1 \) - has token
The initial state of \( p \) is \( s_1 \), the initial state of \( q \) is \( s_0 \) thus the initial global system state is \( \{ p : s_0, q : s_1 \} \), all channels are empty.

Event in the system can be for instance: \( e_0 = < p, s_1, s_0, \text{p has token, c} > \) which means that processor \( p \) sends the message: “pass token” throw the channel \( c \) and the state of \( p \) changes from \( s_1 \) to \( s_0 \).

In figure 2 we can see all the possible global system states, the events that occurred that changed the global states and the function \( \text{Next()} \) calculation for each event.

![Figure 2: Global states, events and Next() calculation for the single token conservation system](image)

The computation of the system in this example was the event sequence: \( (e_0, e_1, e_2, e_3) \). notice that the sequence: \( (e_0, e_2, e_1, e_3) \) for example, can not be a computation of the system.

The algorithm requirements

We require that:

- The snapshot algorithm must run concurrently with the system computation. This means that the scenario were in order to take the snapshot we have to stop the main computation, take the snapshot and then continue with the system computation is unacceptable.

- The snapshot algorithm can not alter the computation in any way.

- Any messages sent for recording purpose must not interfere with the computation of the system.

Now, that we established the “ground rules” for the algorithm we can start thinking on how can we actually take the snapshot and analyze the problems that might occur.

First try and the algorithm challenges

The first try for a snapshot algorithm is simply for each processor and channel to record themselves spontaneously, then use all of the recordings as the system snapshot.

what is the problem with this course of action? The problem is, that we might get a system snapshot that recorded a system state that is illegal for the system. That is, no computation of the system will ever reach that state. To demonstrate this we review figure 3.
In this example we can see the single token conservation system starting a computation and the processors and channels recording spontaneously which leads to a snapshot of an illegal system state - two tokens exist in the snapshot.

Now, let's look at another example, see figure 4.

In this example we look again at the single token conservation system starting a computation and the processors and channels recording spontaneously. Again this leads to a snapshot of an illegal system state - there are no tokens in the snapshot.

**Analysis of the problem:**

Given a processor $p$ and an outgoing channel of it $c$ we will denote:

- $n$ - the number of messages in $c$ when $p$ is recorded
- $n'$ - the number of messages in $c$ when $c$ is recorded

In the first example we had: $n = 0$, $n' = 1$. In the second example we had $n = 1$, $n' = 0$. We conclude from the examples that we must demand from the snapshot that $n = n'$ (since any other case results in an illegal system state of the snapshot). This analysis leads us to the following conditions of the algorithm that will insure a legal system state in the snapshot.
The algorithm conditions

Notations:
for two processors $p, q$ and a channel $c$ between them from $p$ to $q$

- $n$ - the number of messages sent through $c$ before $p$ was recorded
- $n'$ - the number of messages sent through $c$ before $c$ was recorded
- $m$ - the number of messages received from $c$ before $q$ was recorded
- $m'$ - the number of messages received from $c$ before $c$ was recorded

The conditions:

- $n = n'$
- $m = m'$
- $n' \geq m'$
- $n \geq m$
- if $n' = m'$, the recorded state of $c$ must be the empty sequence.
- if $n' > m'$, the recorded state of $c$ must contain the messages: $[\text{tail}]n', \ldots, m'+1[\text{head}]$ (see figure 5).

Figure 5: The messages recorded for a channel in a snapshot

In a less formal way, we want the recorded state of the channel $c$ to be the sequence of messages sent along $c$ before the state of $p$ is recorded, excluding the sequence of messages received along $c$ before the state of $q$ is recorded (see figure 6).

Figure 6: Another view of the messages recorded for a channel in a snapshot

The algorithm

Outline
For two processors $p, q$ and a channel $c$ between them from $p$ to $q$. $p$ will send a special message called a $\text{marker}$ after the $n$-th message it sent (and before sending other message). $q$ will be responsible for recording channel $c$ state. The recorded state will be the messages received by $q$ after $q$ recorded its state and before $q$ received the $\text{marker}$. $q$ will record its state spontaneously or immediately after the $\text{marker}$ is received (that is, before receiving or sending any other messages).
Marker sending and receiving rules:

- Marker-Sending Rule for a Processor $p$:
  - For each channel $c$ directed away from $p$, $p$ sends one marker along $c$ right after $p$ records its state and before $p$ sends further messages along $c$.

- Marker-Receiving Rule for a processor $q$:
  - On receiving a marker along a channel $c$
  - If $q$ has not recorded its state then
    1. $q$ records its state
    2. $q$ records the state of $c$ as the empty sequence
  - Else
    1. $q$ records the state of $c$ as the sequence of messages received along $c$ after $q$’s state was recorded and before $q$ received the marker along $c$.

Some notes about the algorithm

The algorithm can be initiated by one or more processors. Each processor records its state spontaneously (without receiving markers from other processors).

The collection of the snapshot “pieces” from each processor is a topic for a separate discussion. However, one simple possible way to collect the information (if the network is strongly connected) is to find a minimum spanning tree of the graph, each processor will send its own record and the recording received from its children to the parent node, and the root of the tree will collect all the recorded parts and form from them a complete snapshot.

Running example

In the following example, we follow the execution of the algorithm on the single token conservation system and view the snapshot that is produced for the algorithm run on this system and for that computation (see figure 7).

Figure 7: The algorithm running example on the single token conservation system

Termination and correctness of the algorithm

We will now prove that the algorithm is correct - that is, every processor and channel are recorded during the algorithm run and the algorithm terminates. (Later, we will also prove that the snapshot is “useful” in terms of system stability we mentioned earlier in this summary).
Lemma 1:
Given two processors in the system \( p, q \) such that there is a path in the system from \( p \) to \( q \) and \( p \) recorded itself, \( q \) will record itself in finite time.

Proof:
We will prove by induction over the length of the path from \( p \) to \( q \) that \( q \) will record itself in finite time.

*The base case - a path in length 1:*

If \( q \) recorded itself spontaneously then we are done. Let us assume then that \( q \) did not recorded itself spontaneously. The length of the path is 1 so \( q \) is directly connected to \( p \). When \( p \) recorded itself it sent a *marker* in the channel to \( q \) (this is the marker sending rule). The marker will eventually reach \( q \) (according to the distributed model) and when \( q \) receives the marker it will record itself (this is the marker receiving rule). Thus in this case, \( q \) will record itself in finite time.

*Inductive step:*

We assume the lemma is true for a path in length \( k - 1 \) and show that the lemma is true for a path of length \( k \).

If \( q \) recorded itself spontaneously then again we are done. Otherwise, \( q \) is reachable from \( p \), let denote with \( q' \) the last processor in the path from \( p \) to \( q \) before \( q \). The path from \( p \) to \( q' \) is of length \( k - 1 \), so from the induction hypothesis, the processor \( q' \) will record itself in finite time. When \( q' \) will record itself (just like in the base case) it will send a marker to \( q \) and \( q \) in its turn will record itself on receiving the marker. This completes the proof of lemma 1.

Lemma 2:
The algorithm terminates in finite time, with a recording of each processor and channel

Proof:
All the processors will eventually record their state (spontaneously, or because some other processor recorded itself as we know from Lemma 1) this means every processor will send a marker throw all of its outgoing channels (this is true because of the marker sending rule) So, a marker will be sent throw all channels. Once the marker reaches its destination the channel will be recorded by the processor that received the marker (this is true because of the marker receiving rule). The above is true for all channels since all of them had a marker sent throw them. Thus, all the channels are recorded in finite time.

In conclusion, all processors are recorded in finite time, and all the channels are recorded in finite time, thus the algorithm terminates and is correct.

The usability of the snapshot

Lets take a look at another running example of the algorithm, this time on a non-deterministic system.

the non deterministic system properties:

- The system will have the same topology as the single token conservation system meaning it will have two processors: \( p, q \) and two communication channels: \( c, c' \) connecting them.
- \( p \) can be in one of two states: \( \{A, B\} \) and \( q \) can be in of two states: \( \{C, D\} \)
- \( p \) can send the message \( M \) while in state \( A \) and sending the message causes it to move to state \( B \).
- \( p \) can receive the message \( N \) while in state \( B \) and receiving the message causes it to move back to state \( A \).
- \( q \) works symmetrically to \( p \).

A possible computation of the system can be seen in figure 8.
Note that from state $S_0$ (for instance) the event $e_0' = <q, C, D, N, c'>$ could have happened which shows that the system computation is not deterministic.

**The non-deterministic running example**

In the following example, we follow the execution of the algorithm on the non-deterministic system and view the snapshot that is produced for the algorithm run on this system and for that computation (see figure 9).

What is strange about this snapshot? The answer is, that it was not one of the states the system was in during the computation we saw. More generally, the snapshot the algorithm takes is not necessarily a global state the system was in. However, the snapshot is a reachable global state of the system and in addition if the events were to occur in a different order, the snapshot would have been one of the global states reached in that computation. This makes the snapshot consistent and useful when we are trying to decide if a system has reached stability. If a system has reached stability, all of its future possible global states will be true in respect to the stability predicate and that includes the global state the snapshot has recorded.

We will next prove that the global state the snapshot recorded is a legal state of the system.

**Theorem:**

Given:

- $\textbf{seq} = (e_i, i \geq 0)$ a computation of some system.
- $S_i$ the global state of the system before event $e_i$.
- $S_j$ the initial global state of the system.
- $S_k$ the global state of the system when the algorithm terminated $(0 \leq j \leq k)$.
- $S$ the global state the algorithm recorded (the snapshot).
Then there is a computation of the system seq’ that:

- For all \( i, i < j \) or \( i \geq k \), \( e'_i = e_i \).
- For all \( i, i \leq j \) or \( i \geq k \), \( S'_i = S_i \).
- The sub sequence \( (e'_i, j \leq i < k) \) is a permutation of the sub sequence \( (e_i, j \leq i < k) \).
- There exists some \( t, j \leq t \leq k \), such that \( S* = S'_t \).

See figure 10 for an illustration of the above notations.

Definitions:

- **pre-recording event** – an event that occurred in processor \( p \) before \( p \) recorded it’s state.
- **post-recording event** – an event that occurred in processor \( p \) after \( p \) recorded it’s state.

Theorem proof:

First lets note that for event \( e_i \) in seq: if \( i < j \) then \( e_i \) is a pre-recording event and if \( i \geq k \) then \( e_i \) is a post-recording event.

Second lets note that for event \( e_i \) in seq such that \( j < i < k \) the event \( e_{i-1} \) can be a post-recording event and the event \( e_i \) can be a pre-recording event if they occurred in different processors. If they occurred in the same processor and \( e_{i-1} \) is a post-recording event then both must be post-recording events.

**Lemma 3:**

For two given consecutive events \( e_{i-1} = < p, a, b, M, c >, e_i = < q, a', b', M', c' > \) such that \( e_{i-1} \) is a post-recording event and \( e_i \) is a pre-recording event always \( M \neq M' \) and \( c \neq c' \) (or in other words, \( q \) can not be receiving the message \( p \) sent).

**Proof:**

\( e_{i-1} \) is a post-recording event which means that a marker was sent in \( c \) before \( M \) was sent. The same marker was received by \( q \) before \( M \) reached it. When \( q \) received the marker it recorded itself so if \( e_i = < q, a', b', M, c > \) it can only be a post-recording event in contradiction to the fact that \( e_i \) is a pre-recording event.

**Conclusion:**

The two events \( e_{i-1} \) and \( e_i \) are independent of each other which means that we can swap their order in the computation seq. The new computation \( \ldots, e_{i-2}, e_{i-1}, e_i, \ldots \) will end with the same global state as the original computation \( \ldots, e_{i-2}, e_{i-1}, e_i, \ldots \) (see figure 11 for an illustration of this conclusion).
Let $seq'$ be a computation were every post-recording event that occur right before a pre-recording event are swapped. We repeat the swapping until $seq'$ has all pre-recording events before post-recording events. Note that:

- $seq'$ is a computation of the system.
- For all $i$, $i < j$ or $i \geq k$, $e'_i = e_i$
- For all $i$, $i \leq j$ or $i \geq k$, $S'_i = S_i$

Now, let's look at the global system state after the last pre-recording event and before the first post-recording event. We will denote this state as $S_t$ ($j \leq t \leq k$).

**Processors state in $S_t$:** For some processor $p$, let us assume the last state $p$ was in before recording is $a$ (that means $p$ recorded $a$ as its state). In the global state $S_t$ we will see that $p$ is in state $a$. In the snapshot $S*$ we also see that the state of $p$ is $a$ (because $p$ recorded $a$).

Conclusion: the state of each processor in $S_t$ is the same as in $S*$.

**Channels state in $S_t$:** For some channel $c$ from $p$ to $q$, in $S_t$ the messages in $c$ are the ones $p$ sent before sending a marker in $c$ (before $p$ recorded itself) without the messages $q$ received before recording itself. In the snapshot $S*$, $c$ contains all the messages $q$ received in $c$ after it recorded itself and before it received a marker in $c$.

Conclusion: the messages in $c$ in the global state $S_t$ and in the snapshot $S*$ are the same.

**Theorem proof summary:** We saw that there is a computation of the system - $seq'$ (that we have shown above) such that

- For all $i$, $i < j$ or $i \geq k$, $e'_i = e_i$.
- For all $i$, $i \leq j$ or $i \geq k$, $S'_i = S_i$.
- The sub sequence $(e'_i, j \leq i < k)$ is a permutation of the sub sequence $(e_i, j \leq i < k)$.
- There exists some $t$, $j \leq t \leq k$, such that $S_* = S'_t$, (this is the global system state after the last pre-recording event and before the first post-recording event).

This concludes the theorem proof. See figure 12 for an example of permuting a computation.

---

**Figure 12: An example of permuting a computation**

![Image of a computational example](image12.png)

The non-deterministic example computation we saw:

<table>
<thead>
<tr>
<th>$S_0$</th>
<th>$e_0 = p, A, B, M, c$</th>
<th>Post-recording</th>
<th>Next($S_0, e_0$) = $S_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$e_1 = q, C, D, N, c$</td>
<td>Pre-recording</td>
<td>Next($S_1, e_1$) = $S_2$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$e_2 = p, B, A, N, c$</td>
<td>Post-recording</td>
<td>Next($S_2, e_2$) = $S_3$</td>
</tr>
</tbody>
</table>

The recorded global state:

- $p$  
- $c$  
- $q$  
- $c'$

A empty D N

After swapping the events so all pre-recordings will precede post-recordings we get the next computation:

<table>
<thead>
<tr>
<th>$S'_1$</th>
<th>$e'_1 = q, C, D, N, c$</th>
<th>Pre-recording</th>
<th>Next($S'_1, e'_1$) = $S'_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S'_2$</td>
<td>$e'_2 = p, A, B, M, c$</td>
<td>Post-recording</td>
<td>Next($S'_2, e'_2$) = $S'_3$</td>
</tr>
</tbody>
</table>

The global state $S'_t$ of this new computation is exactly the snapshot of the original computation.
Usability example

After showing that the snapshot is usable, we will now demonstrate an algorithm for checking if a system is stable using our snapshot.

Stability detection algorithm:

*Input*: a stable property predicate function $f$

*Output*: a boolean value *true* if we the system is stable, *false* if we cannot yet determine system stability.

*Algorithm:*

1. record a global state $S*$;
2. $bRes = f (S*)$;
3. return $bRes$;

**Correctness:**

If $bRes$ is *true* then because $S*$ is reachable from the initial state, the final state is reachable from $S*$ and the stability property insures that $f (S) \Rightarrow f(S')$ for all $S'$ reachable from $S$, we know the final state of the computation has reached stability, thus the whole system is stable.

If $bRes$ is *false* then the predicate doesn’t hold for $S*$, but, it may hold for the final state of the computation (we don’t know what is the value of the final state just that it exists). So in this case we don’t know for sure if the system has reached stability or not.

Summary and conclusions

We saw an algorithm for a snapshot in a distributed system. We saw the algorithm presented terminates with a recording of every processor and channel of the system and we saw the snapshot the recordings create is a legal system state the system could have reached. Finally we demonstrated how to use the snapshot in a stability checking algorithm. With that we conclude this paper summary.

References
