Advanced Blockchain Engineering (ASCI a27)

Chapter 5. Stabilization
- Concepts and Definitions
- Mutual Exclusion
- Protocol Combination
- Datalink Protocols

Dick H.J. Epema
Distributed Systems Group

April 2018
Stabilization (1/3)

- Computer systems can exhibit **faults**:
  - memory may be corrupted
  - messages may not be (correctly) received

- Faults can be **permanent** or **transient**:
  - a processor may crash completely
  - a processor may restart in some unknown state

- Dealing with faults can be done **explicitly**:
  - detect and correct errors in data
  - resend a message when no acknowledgment is received
  - use a consensus protocol

- Or **implicitly**:
  - have algorithms converge to a correct state as a side effect of their operation
Stabilization (2/3)

• **A stabilizing algorithm:**
  - deals with transient faults in an implicit way by **converging** to the desired behavior
  - can be **started in an arbitrary state** without the need for a global (synchronized) system initialization
  - can deal with **system reconfigurations** without the need for a complete system reset

• **Real-world metaphor:**
  - an orchestra plays outdoors without a conductor
  - the pages of the score can be turned by the wind
  - the players try to **(re-)synchronize** their playing
  - each player restarts at the **lowest-numbered page** he hears being played from around him
  - the score should be long enough
Stabilization (3/3)

- Old term: self-stabilizing algorithms
- Stabilizing algorithms are used for non-terminating control and coordination tasks:
  - mutual exclusion
  - election
  - maintaining routing tables
  - maintaining a spanning tree
Types of transient faults

• A transient fault is an event that
  o corrupts the state of a program (including its PC)
  o does not corrupt the program itself

• Possible causes of such faults:
  o memory crashes
  o transmission errors
  o process failure and recovery
  o inconsistent initializations

• Reasons for assuming programs to remain correct:
  o programs may reside in ROM (e.g., in embedded systems)
  o programs may be reloaded from disk (e.g., periodically)
  o program code is in general immutable, so can be protected by redundancy
Definitions

- For a DA, a set of **legal** (legitimate, safe) states is defined (possibly by a **predicate**)
- For instance, in a mutual-exclusion algorithm:
  \[ \#\{i \mid \text{in}_{CS_i}=1\} \leq 1 \]
- Assume \( P \) to be a **stable predicate**
- **Convergence**: A DA \( P \)-stabilizes if in any (fair) execution, from some point onwards, the system is in a state satisfying \( P \)
The first stabilizing algorithm (1/2)

• Due to E.W. Dijkstra, 1974
• Solves the mutual-exclusion problem in a unidirectional ring of $N$ processes
• Process $P_i$ sends to process $P_{i+1}$ and receives from process $P_{i-1}$ (process ids modulo $N$)
• Process $P_i$ has a single integer variable $x_i$, modulo some $K$
• Assume a shared-memory model with composite atomicity (reading and writing in one atomic step)
The first stabilizing algorithm (2/2)

- An **atomic step in process** $P_i$, $i \neq 0$ (copy value of predecessor if **unequal** to predecessor):

  
  \[
  \text{if } (x_i \neq x_{i-1}) \text{ then } \quad x_i := x_{i-1}
  \]

- An **atomic step in process** $P_0$ (the **special process**, increment value if **equal** to predecessor):

  
  \[
  \text{if } (x_0 = x_{N-1}) \text{ then } \quad x_0 := (x_0 + 1) \mod K
  \]
Example 1 (a)

<table>
<thead>
<tr>
<th>x_0</th>
<th>x_1</th>
<th>x_2</th>
<th>...</th>
<th>x_{N-2}</th>
<th>x_{N-1}</th>
<th>step by</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>P_0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>P_1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>P_2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>0</td>
<td>0</td>
<td>P_3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>0</td>
<td>P_{N-1}</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>...</td>
<td>1</td>
<td>1</td>
<td>P_0</td>
</tr>
</tbody>
</table>

- Always, exactly one process can take a step
- The opportunity for taking a step (that changes the state) travels around the ring
Example 1 (b)
### Example 2

<table>
<thead>
<tr>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_{N-2}$</th>
<th>$x_{N-1}$</th>
<th>step by</th>
</tr>
</thead>
<tbody>
<tr>
<td>N-1</td>
<td>N-2</td>
<td>N-3</td>
<td>...</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>N-1</td>
<td>N-1</td>
<td>N-3</td>
<td>...</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>N-1</td>
<td>N-1</td>
<td>N-1</td>
<td>...</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>N-1</td>
<td>N-1</td>
<td>N-1</td>
<td>...</td>
<td>N-1</td>
<td>0</td>
</tr>
<tr>
<td>N-1</td>
<td>N-1</td>
<td>N-1</td>
<td>...</td>
<td>N-1</td>
<td>N-1</td>
</tr>
</tbody>
</table>

- Many processes are initially enabled
- Processes copy value along the ring
- In the end, only one process enabled
- Here $N-1$ steps are needed
Questions

Q: What is the maximum number of steps before only one process is enabled in Example 2?
A: \((N-1) + (N-2) + \ldots + 2 + 1 = \frac{N(N-1)}{2}\)

Q: What are the legal states in the ring?
A: \(k, k, \ldots, k, k-1, k-1, \ldots, k-1\)
Correctness (1/3)

- **Idea: missing-label concept:**
  - show that eventually, $P_0$ introduces a **new value** into the system
  - after $P_0$ has done so, it is only enabled again when all process have this value, which is a legal state

- **Correctness proof:**
  - all the $P_i$ with $i>0$ **only copy values**
  - $P_0$ is the only process capable of **introducing a new value** into the system
  - $x_0$ cycles through the values $0,1,2,...,K-1$
  - after at most $1+2+3+...+(N-1)=N(N-1)/2$ steps, $P_0$ is enabled again
Correctness (2/3)

- **Correctness proof (cont’d):**
  - if $K>N$, at least one value in $0,1,...,K-1$ is not in the ring, so at some point $P_0$ introduces a new value
  - if $K=N$, if all values are present in the ring, $x_0$ has a unique value, so stabilization is also guaranteed
  - if $K=N-1$, then still stabilization can still be shown
Correctness (3/3)

- **Counter example** for $N=5$, $K=3$:

  - Exercise: generalize this example for any $N>5$, $K=N-2$
**Remarks**

- If the number of processes in the ring is **not a prime**, then a **special process** is needed.
- Shared memory with **read/write atomicity**: in one atomic communication step only reading or writing (also called the link-register model).
- Under read/write atomicity:
  - there are **2N** variables (one in every process, and one for communication between successive processes)
  - then stabilization is possible if **K>2N-2**
- Stabilization in a distributed-memory model requires a **stabilizing algorithm for a network link**.
Fair protocol composition (1/2)

- **Idea:** split up a stabilizing algorithm into two or more simpler stabilizing algorithms ("levels")

- **Use:**
  - simpler design process
  - reuse of components
  - the possibility of *algorithm conversions* between different system models

- **Example:**
  - a stabilizing datalink protocol in an asynchronous message-passing system
Fair protocol composition (2/2)

- Task $T_1$ with stabilizing protocol $P_1$, with state $A$
- Task $T_2$ with stabilizing protocol $P_2$ with state $AxB$
- State transition functions:
  - for $P_1$: $f: A \rightarrow A \quad (a,b) \rightarrow (f(a),b)$
  - for $P_2$: $g: AxB \rightarrow B \quad (a,b) \rightarrow (a,g(a,b))$
A stabilizing datalink algorithm (1/3)

- A **stop-and-wait** algorithm
- Sender and receiver maintain counters
- Receiving an ack in the **sender**:
  
  ```
  upon reception of (r_counter) do
    if (r_counter≥s_counter) then
      s_counter := s_counter+1
      send(message[s_counter];s_counter)
  /* when ack received */
  /* if ack high enough */
  /* increment send counter */
  /* and send next message */
  ```
A stabilizing datalink algorithm (2/3)

- Receiving a message in the **receiver**:
  
  ```
  upon reception of (message;s_counter) do
  if (s_counter≠r_counter) then  /* when new message */
    process message
    r_counter := s_counter  /* adapt counter and */
    send(r_counter)  /* send ack back */
  ```

- Timeout in **sender**:
  
  ```
  upon timeout send(message[s_counter];s_counter)
  ```
A stabilizing datalink algorithm (3/3)

- Main idea of stabilization:
  - system is in a legal state if all counters (in sender, in receiver, and in messages) are equal
  - the sender will always introduce new (higher) values for its counter into the system
  - then, at some later time, all counters will be equal
  - resembles missing-label proof of Dijkstra’s algorithm
A datalink algorithm (1/5)

• A sliding-window protocol

• Variables in the sender S:
  - w: the window size
  - ns: the number of the next message to send
  - na: the number of the first non-acknowledged message

• Variable in the receiver R:
  - nr: the number of the next message to be received
A datalink algorithm (2/5): full window

• Let $w=3$
• Let
  • $ns=7$
  • $na=nr=4$
• Sender has to wait because window is full: $ns=na+w$
• Channel SR contains messages 4,5,6
A datalink algorithm (3/5): empty window

- Let \( w = 3 \)
- Let
  - \( ns = na = nr = 7 \)
- Channel SR is empty because all messages sent have been ack-ed

\[
\begin{align*}
\text{new messages} & \quad 7 \quad 6 \quad 5 \quad 4 \\
\text{sent and ack-ed} & \quad \text{not yet sent} \\
\text{ns} &= \text{na} = \text{nr} = 7
\end{align*}
\]
A datalink algorithm (4/5): acks

- When R receives a message with number \( n \), it:
  - first sets \( nr \) to \( n+1 \)
  - and then sends a message \((\text{ack};nr)\)
- So \((\text{ack},i)\) acknowledges the receipt of messages \(1,2,\ldots,i-1\)
- Upon reception of \((\text{ack},i)\), S sets \( na=i \)
A datalink algorithm (5/5)

- **Stable predicate $P$:**
  
  \[(na \leq nr) \land (nr \leq ns) \land (ns \leq na + w)\]

  1.a-c

  and

  (for each (message; $i$) in channel $SR$, $i < ns$)  2

  and

  (for each (ack; $i$) in channel $RS$, $i \leq nr$)  3

- **Explanation:**
  
  1.a: no message acknowledged that has not yet been received

  1.b: no message received that has not been sent

  1.c: not more than $w$ outstanding messages

  2: no message on channel $SR$ that has not been sent

  3: no message acknowledged that has not been received
The protocol (1/3): the sender

I. Sending a message
   if (ns<na+w) then /* window not full */
   send(message[ns];ns)
   ns := ns+1

II. Receiving an acknowledgment in the sender
   upon receipt of (ack;i) do
      if (i>na) then na:=i /* adapt na */

III. Timeout in the sender
   upon timeout do
      if (ns>na) then /* resend un-ack-ed messages */
      for i=na,na+1,...,ns-1 do
         send(message[i];i)
The protocol (2/3): the receiver

IV. Receiving a message

upon reception of (message;i) do

if (i=nr) then /* expected message */
  nr:=nr+1
send(ack,nr) /* send an ack with nr anyway */
The protocol (3/3): no stabilization

- This protocol does **not stabilize**
- Assume **w=1**
- Consider the state
  - **nr=3** (R is still waiting for message 3)
  - **na=4** (S waits for the ack for message 4)
  - **ns=5** (S wants to send message 5)
  - both channels empty

- Message 3 is lost
- Sent but not ack-ed
- Violates predicate
Stabilizing version (1/4)

- **S** catches up with **R**, when **R** is ahead of **S**:
  - when **S** receives (ack,i) with i>ns, it sets na and ns to i  (jump ahead)

- Messages carry **two sequence numbers i and j**:
  - i as before
  - j is the value of na at the time of sending the message
  - with j, **R** gets to know what **S** thinks **R** has received

- **R** catches up with **S**, when **S** is ahead of **R**:
  - when **R** receives message[i,j] with j>nr, it sets nr to j (so it jumps ahead, and misses messages)
Stabilizing version (2/4): the sender

I. Sending a message
   
   if ((na≤ns) and (ns<na+w)) then /* if there is something to send */
   send(message[ns];ns,na) /* and window not full */
   ns := ns+1 /* now ns larger than j in message */

II. Receiving an acknowledgment in the sender
    upon receipt of (ack;i) do
       if ((i>na) and i≤ns)) then na:=i /* expected */
       else
          if ((i>na) and (i>ns)) then na, ns:=i, i /* R is ahead */
          /* S adapts */
Stabilizing version (3/4): the sender

III. Timeout in the sender

upon timeout do

if (na≠ns) then /* otherwise empty window */
  if (na>ns) then ns:=na /* wrong situation: reset (window empty) */
  if (ns>na+w) then na:=ns /* wrong situation: reset (window empty) */
for i=na,na+1,...,ns-1 do /* resend unacknowledged messages */
  send(message[i];i,na)

|   na+w | ns | na  |
Stabilizing version (4/4): the receiver

IV. Receiving a message

upon reception of (message;i,j) do

if ((i=nr) and (j≤nr)) then /* expected message and not */
    nr:=nr+1 /* too much acked */
else
    if (j>nr) then nr:=j /* adapt to S and jump ahead */
    send(ack,nr) /* send an ack anyway */
Stable predicate

\(((na \leq nr) \land (nr \leq ns) \land (ns \leq na + w))\)

and

(for each (message; i, j) in channel SR,
  \(i < ns \land j \leq nr \land j < ns\))

and

1 and 2

(for each (ack; i) in channel RS, \(i \leq nr\))

**New part** says:

1. \(na\) in S does not exceed \(nr\) in R, that is, not more acknowledged at S than received by R

2. \(na\) at the time of sending in S is smaller than the current value of \(ns\), as \(ns\) is incremented after sending a message

**Exercise:** show that the case where the initial protocol went wrong is now handled correctly
Remarks

• One can show that stabilizing datalink protocols:
  o do not terminate
  o need unbounded counters
  o need timeout actions