Formal Specification of Distributed Systems: Communicating Sequential Processes (CSP)

Nicola Dragoni
Embedded Systems Engineering
DTU Informatics

Protocols Specification
Simple ACK/NACK Protocol

- $M$ = domain of correct messages
- $M'$ = domain of corrupted messages
- $M \cap M' = \{\}$
Polling

- In the previous simple ACK/NACK protocol:
  - it is the sender who takes the initiative for sending a message
  - the receiver merely responds to this.

- Effectively, this obliges the receiver to be able to receive data at any time after it has sent an acknowledgment.
Polling

• In the previous simple ACK/NACK protocol:
  
  ‣ it is the sender who takes the initiative for sending a message
  
  ‣ the receiver merely responds to this.

• Effectively, this obliges the receiver to be able to receive data at any time after it has send an acknowledgment.

• Alternative strategy (POLLING):
  
  ‣ the receiver explicitly takes the initiative, requesting data when it is able to receive them.
Simple Polling Protocol

• Receiver has initiative.

• Messages:
  - POLL: request to send data
  - REPT: request to repeat transmission of data received with errors
ACK/NACK Problem

msg → Sender → msg → Receiver

deadlock!
ACK/NACK + TIMEOUT

- Deadlock caused by loss of the acknowledgment message.
- Corrected by retransmission after a certain time with no acknowledgment.

- Timer in CSP
- ACK/NACK protocol with TIMEOUT
ACK/NACK + TIMEOUT - Duplication Problem

Consider the following situation:

- the receiver receives a correct message via its channel left and then sends a positive acknowledgment
- this acknowledgment message gets lost
- the sender will eventually time out, and retransmit the same message to the receiver
- so the receiver receives the message twice and passes it on to the user (via SAPB) twice.
ACK/NACK + TIMEOUT - Duplication Problem

Sender

msg

duplication!

Receiver

msg

ack

timeout

msg

msg
Possible Solution: Numbering Scheme

• Introducing a numbering scheme for the messages: duplicated messages can be filtered off by the receiver before messages are passed to the user.

Sender

msg1

timeout

msg2

Receiver

msg1

ack

msg1

ack

msg2

ack

not sent

msg1

msg2

msg1

not sent

msg1
Exercise: ACK/NACK + TIMEOUT + NUMBERING SCHEME

• Write a CSP specification of an ACK/NACK protocol able to handle the following failures:

  1. deadlock caused by the loss of the acknowledgment message

  2. duplication of messages sent to the user

HOMEWORK
PAR Protocols

• We can also remove the NACK type of acknowledgment. Why?

  ‣ When a timeout mechanism is used, negative acknowledgments only have an effect on the response time of the protocol, since they can be used to provoke retransmission before the timeout period runs out.

  ‣ Negative acknowledgments do not affect the logical properties of the protocol in any way.
PAR Protocols

- We can also remove the NACK type of acknowledgment. Why?
  - When a timeout mechanism is used, negative acknowledgments only have an effect on the response time of the protocol, since they can be used to provoke retransmission before the timeout period runs out.
  - Negative acknowledgments do not affect the logical properties of the protocol in any way.

- Protocols with:
  - only positive acknowledgments +
  - using a timeout mechanism to control retransmission

are often called Positive Acknowledge and Retransmission (PAR) protocols.
PAR Protocol (ACK + TIMEOUT + NUMBERING SCHEME)

- PAR protocol with
  - ACK messages
  - TIMEOUT
  - Sequence numbers in data
1. Extend the following polling protocol with sequence numbers and timeout.

\[
\begin{align*}
\text{Sender} & \overset{\text{def}}{=} Q[a] \\
Q[x : M] & \overset{\text{def}}{=} (\text{right?}y : \{\text{POLL}\} \rightarrow SAPA?x : M \rightarrow \text{right!}x \rightarrow Q[x] \\
& \quad \Box \text{right?}y : \{\text{REPT}\} \rightarrow \text{right!}x \rightarrow Q[x]) \\
\text{Receiver} & \overset{\text{def}}{=} (\text{left!POLL} \rightarrow R) \\
R & \overset{\text{def}}{=} (\text{left?}x : M \rightarrow SAPB!x \rightarrow \text{Receiver} \\
& \quad \Box \text{left?}y : M' \rightarrow \text{left!REPT} \rightarrow R)
\end{align*}
\]

- N.B.: In this case, the timeout function would most naturally be incorporated in the receiver, which would poll the sender again if it did not receive a message within a suitable period of time.

2. Analyze your proposal to see which problems the protocol might still have.
PAR Protocol - Problem

I got an ack for 2, so I send 3

I’m waiting for 2 (or 1)
PAR Protocol - Problem

Sender

msg1

msg2

I got an ack for 2, so I send 3

Receiver

msg1

msg2

ack

msg1

ack

msg3

I’m waiting for 2 (or 1)

The acknowledgements are anonymous!
Problem: Anonymous ACK

- All protocols we have seen so far have the anonymous acknowledgement problem because they rely on anonymous messages.
- This reflects a general problem in distributed systems: the cooperating parties do not in general know what their collective global state is.
- Parties have to make decisions on the basis of
  - whatever information they locally have available or
  - the information their cooperators have sent them.
- ACK messages:
  - just tell the sender that the other party has received the data which came in the right order
  - the sender has no means of knowing exactly which data is referred to.
Solution: Sequence Numbers in ACKs

• We include an identification on the acknowledgments, indicating the sequence number of the latest correctly received data.

• Sender:
  ‣ repeats message with number $n$ until it receives an acknowledgment explicitly denoting $n$.

• Receiver:
  ‣ replies to each correct incoming data with an acknowledgment that includes the sequence number of the last correctly received message (which of course may be the message just received or a previous one).
Example: PAR Protocol + NUMBERED ACK

• $E$ denotes the domain of corrupted ack messages
PAR Protocol + NUMBERED ACK

• The ack message now consists of the NUMBER OF THE LATEST CORRECTLY RECEIVED data message.

• $E$ denotes the domain of corrupted ack messages.

• Protocol now gives both parties sufficient knowledge of what is happening, so it protects against
  ‣ loss
  ‣ duplication
  ‣ corruption

of both data messages and ack messages.
The ack message now consists of the **NUMBER OF THE LATEST CORRECTLY RECEIVED** data message.

- $E$ denotes the **domain of corrupted ack messages**.

Protocol now gives both parties sufficient knowledge of what is happening, so it protects against

- **loss**
- **duplication**
- **corruption**

But it can still fail. How?

of both **data messages** and **ack messages**.
Sequence Numbers?

- **Simple idea**: Sequence numbers are successive natural numbers 0, 1, 2, 3, ...

- **Problem**: Only a finite number can be represented in a real message.
Sequence Numbers?

• **Simple idea:** Sequence numbers are successive natural numbers 0, 1, 2, 3, ...

• **Problem:** Only a finite number can be represented in a real message.

• **New idea:** If acknowledgment is received within relatively short time, it is only necessary to count modulo some small value \( S_{mod} \), so

\[
\text{succ}(n) \overset{\text{def}}{=} (n + 1) \mod S_{mod}
\]

• **Example** [PAR protocol with numbered ACK]: Sender always waits for positive ACK for latest transmitted message before using next sequence number. OK to count modulo 2 ("Alternating Bit Protocol").
Sequence Numbers?

- **Simple idea**: Sequence numbers are successive natural numbers 0, 1, 2, 3, ...

- **Problem**: Only a finite number can be represented in a real message.

- **New idea**: If acknowledgment is received within relatively short time, it is only necessary to count *modulo* some small value $S_{mod}$, so

  $$\text{succ}(n) \overset{\text{def}}{=} (n + 1) \mod S_{mod}$$

- **Example** [PAR protocol with numbered ACK]: Sender always waits for positive ACK for latest transmitted message before using next sequence number. OK to count modulo 2 ("Alternating Bit Protocol").

- If more messages can be outstanding (sent but not acknowledged), $S_{mod}$ must be larger.

- **ESSENTIAL RULE**: messages with number $n$ must be guaranteed to be "dead" before $n$ is re-used.
Floating Corpses

• Imagine a system where **msgs can get lost for a considerable period of time.**

• In our protocols:
  
  ‣ The sender eventually times out, declares the messages “dead”, and retransmits them.
  
  ‣ The receiver accepts the retransmitted messages.

• All seems well!!
Floating Corpses

• Imagine a system where *msgs can get lost for a considerable period of time.*

• In our protocols:
  ‣ The sender eventually times out, declares the messages “dead”, and retransmits them.
  ‣ The receiver accepts the retransmitted messages.

• All seems well!!

• But at this moment *the corpses come floating up to the top of the service, as it were, and arrive at the receiver.*

• Total confusion arises, as most protocols are unable to counteract this form for masquerading.
Class of Error: Masquerading

- **Masquerading**: introduction by the underlying service (channel) of false messages which look as though they are correct ones.
  
  ‣ For instance: because they have appropriate sequence numbers and belong to the set of correct messages.
Class of Error: Masquerading

- **Masquerading**: introduction by the underlying service (channel) of *false messages* which look as though they are correct ones.

  - For instance: because they have appropriate sequence numbers and belong to the set of correct messages.

- Possible solutions?

  - Never re-use sequence number! Not realistic...

  - Use ENORMOUS sequence number space! After a crash it is extremely difficult to guarantee that we can remember where we got in the sequence numbers.

  - Explicit limits to message lifetime! Several techniques are possible. In practice, combinations of these techniques are often used.
Exchange of State Information

• Can be necessary, for example:
  ‣ To agree on an initial state.
  ‣ To indicate a change of state.
  ‣ To set up or break a connection.
  ‣ To perform an atomic action.

• Reliable exchange requires at least exchanging a message in each direction (Confirmed Exchange).

• Often depicted by

TIME-SEQUENCE DIAGRAM:

- Request
- Indication
- Response
- Confirm
- Time
Two-Way Exchange (or Handshake) Protocol

<table>
<thead>
<tr>
<th>Sender</th>
<th>( \text{def} (S \parallel \text{Timer}) \setminus {\text{up}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( \text{def} (SAPA?r : \text{req} \rightarrow \text{right}!r \rightarrow \text{up}!\text{SET} \rightarrow SR) )</td>
</tr>
<tr>
<td>( SR )</td>
<td>( \text{def} (\text{right}?c : \text{accept} \rightarrow \text{up}!\text{RESET} \rightarrow \text{SAPA}!c \rightarrow (\ldots) )</td>
</tr>
<tr>
<td></td>
<td>( \square \text{right}?a : \text{refuse} \rightarrow \text{up}!\text{RESET} \rightarrow \text{SAPA}!a \rightarrow S )</td>
</tr>
<tr>
<td></td>
<td>( \square \text{up}?t : {\text{TIMEOUT}} \rightarrow \text{SAPA}!\text{ref} \rightarrow S )</td>
</tr>
<tr>
<td>Receiver</td>
<td>( \text{def} (R \parallel \text{Timer}) \setminus {\text{up}} )</td>
</tr>
<tr>
<td>( R )</td>
<td>( \text{def} (\text{left}?r : \text{req} \rightarrow SAPB!r \rightarrow \text{up}!\text{SET} \rightarrow RR) )</td>
</tr>
<tr>
<td>( RR )</td>
<td>( \text{def} (SAPB?c : \text{accept} \rightarrow \text{up}!\text{RESET} \rightarrow \text{left}!c \rightarrow (\ldots) )</td>
</tr>
<tr>
<td></td>
<td>( \square \text{SAPB}?a : \text{refuse} \rightarrow \text{up}!\text{RESET} \rightarrow \text{left}!a \rightarrow R )</td>
</tr>
<tr>
<td></td>
<td>( \square \text{up}?t : {\text{TIMEOUT}} \rightarrow \text{SAPB}!\text{ref} \rightarrow \text{left}!\text{ref} \rightarrow R )</td>
</tr>
</tbody>
</table>

- **Domains:** \text{req}: requests; \text{accept}: positive replies; \text{refuse}: negative replies; \text{ref}: refuse: \textit{internal} message indicating refusal.
- **At** (\ldots), both parties are sufficiently finished to go on with the next part of their tasks.
Exchanges in the Presence of Errors

• We might use the same techniques adopted before (i.e., retransmission, sequence numbers in data and acknowledgments) but...

... how to avoid the FLOATING CORPSES?
Exchanges in the Presence of Errors

• We might use the same techniques adopted before (i.e., retransmission, sequence numbers in data and acknowledgments) but...

  ... how to avoid the FLOATING CORPSES?

• It is not always possible to add sequence numbers to messages used for administrative purposes (for instance, actually establishing connection).

  ‣ The initial sequence number for messages is one of the components of the global state which we wish to establish!

• So we must find some other information which can be exchanged and which will enable us to distinguish false messages from genuine ones during connection establishment.

• In particular, we need another exchange: three-way handshake.
Three-Way Handshake... in a Nutshell

• Used for the connection establishment phase of the Internet TCP Transport layer protocol.

• More generally, the protocol finds uses in all situations where a confirmed service is required over an unreliable underlying service.

• General scheme:
  
  ‣ the initiating protocol entity sends a request message carrying an arbitrary value \(x\)
  
  ‣ the responding entity replies with a response message bearing \((x, y)\)
  
  ‣ the initiating entity repeats this message as an extra confirmation.
Analogy: Exchange of Letters

• An analogy is the use of “our reference” and “your reference” fields in an exchange of letters.

  ▸ If you get a letter with an unknown reference on it, you throw it straight in the wastebin.
Analogy: Exchange of Letters

• An analogy is the use of “our reference” and “your reference” fields in an exchange of letters.

  ‣ If you get a letter with an unknown reference on it, you throw it straight in the wastebin.

• Normal run of the protocol:

\[
\begin{align*}
A & \quad B \\
\rightarrow & \quad < \text{req, } \text{ourref} = x > \quad \rightarrow \quad \text{A initiates.} \\
\leftarrow & \quad < \text{accept, } \text{ourref} = y, \text{ yourref} = x > \quad \leftarrow \quad \text{B responds.} \\
\rightarrow & \quad < \text{check, } \text{ourref} = x, \text{ yourref} = y > \quad \rightarrow \quad \text{A confirms.}
\end{align*}
\]
Three-Way Handshake... in CSP

\[
\begin{align*}
    \text{Sender} & \overset{\text{def}}{=} (S \parallel \text{Timer}) \setminus \{up\} \\
    S & \overset{\text{def}}{=} (SAPA?r : req \rightarrow right!(x, r) \rightarrow \text{up!set} \rightarrow SR[x]) \\
    SR[x : tok] & \overset{\text{def}}{=} (right?(p, q : tok, c : accept) \rightarrow \text{up!reset} \rightarrow \\
        (\text{if} \ (p = x) \\
        \text{then} \ (right!(p, q, check) \rightarrow SAPA!c \rightarrow \ldots) \\
        \text{else} \ (SAPA!ref \rightarrow S)) \\
    \square right?(p, q : tok, a : refuse) & \rightarrow \text{up!reset} \rightarrow \\
        (\text{if} \ (p = x) \\
        \text{then} \ (SAPA!a \rightarrow S) \\
        \text{else} \ SR[x] \)) \\
    \square up?t : \{\text{timeout}\} & \rightarrow \text{SAPA!ref} \rightarrow S)
\end{align*}
\]
Three-Way Handshake... in CSP

\[
\begin{align*}
    \text{Receiver} & \; \overset{\text{def}}{=} (R \ || \ \text{Timer}) \ \setminus \ \{up\} \\
    R & \; \overset{\text{def}}{=} (left?(x : \text{tok}, r : \text{req}) \rightarrow SAPB!r \rightarrow up!\text{SET} \rightarrow RR[x]) \\
    RR[x : \text{tok}] & \; \overset{\text{def}}{=} (SAPB?c : \text{accept} \rightarrow left!(x, y, c) \rightarrow RC[x, y] \\
    & \quad \quad \boxed{SAPB?a : \text{refuse} \rightarrow left!(x, y, a) \rightarrow R} \\
    & \quad \quad \boxed{up?t : \{\text{TIMEOUT}\} \rightarrow left!(x, y, ref) \rightarrow SAPB!ref \rightarrow R}) \\
    RC[x, y : \text{tok}] & \; \overset{\text{def}}{=} (left?(p, q : \text{tok}, c : \text{check}) \rightarrow \\
    & \quad \quad \quad \text{(if } (p = x) \ \& \ (q = y) \text{)} \\
    & \quad \quad \quad \text{then } (up!\text{RESET} \rightarrow \ldots) \\
    & \quad \quad \quad \text{else } RC[x, y] \ ) \\
    & \quad \quad \boxed{up?t : \{\text{TIMEOUT}\} \rightarrow SAPB!ref \rightarrow R})
\end{align*}
\]
What Happens with Floating Corps?

- B responds to a false request message
- A is unable to match B’s reference x to any exchange i which A is currently taking part

\[ A \]
\[
\ldots \quad <\text{req}, \quad \text{ourref} = x > \quad \rightarrow \quad \text{delayed req-PDU} \\
\leftarrow <\text{accept}, \quad \text{ourref} = y, \quad \text{yourref} = x > \quad \leftarrow \\
\]

\[ B \]

B responds.

A gives up.

(B times out.)

\[ \text{===> A gives up and (in our version of the protocol, see next slides) B subsequently times out and therefore also gives up.} \]
What Happens with Floating Corps?

- B responds to a false request message
- but when it receives the false check message from A it finds an incorrect reference $z$ instead of the value $y$ which it itself had generated

$\Rightarrow$ A and B give up without timeout.
Exercise: 3-Way Handshake

• The protocol should survive receipt of out-dated request/response/check messages.

• Analyze the protocol to check whether or not this is really true.

• And:

Could the protocol still fail in some other situation?