Group Initialization

0 Introduction

This note serves to guide an already ongoing discussion; as such this note is not entirely self-contained. The problem to be discussed here is a kind of initialization protocol, under rather weak assumptions.

We consider a (finite) collection of *processes* connected by some communication medium. Via this medium the processes can send *messages* to one another; message transmission is *asynchronous* and is subjected to two restrictions.

First, in order to be able to receive messages a process must have performed some initialization first. The details of this are irrelevant to us, but the essence is that a process must have completed the transition from its *initial* state to a state that I will call *susceptible* here. More precisely, if a process is susceptible at the moment a message is sent (by another process), then arrival of the message at that process is guaranteed. If a process becomes susceptible *after* a message has been sent this message either *may or may not* arrive at this process.

Second, the medium allows a single message to be sent to all susceptible processes; this is called a *broadcast* message. In addition, the medium allows a message to be sent to a single, explicitly identified process; this is called a *direct* message. For this purpose, I assume the processes to be identified by (unique) *names*. To send a direct message the sending process must have the name of the receiving process at its disposal.

Finally, I assume that every message carries the name of its sender, even if the message is otherwise empty.

* * *

Initially, all processes are in their initial states and they do not have the names of the other processes from the collection. The problem to be solved is the construction of an initialization protocol such that, upon its completion, every process is susceptible and every process has the names of all processes from the collection. Here "has the name of ..." is short for "has some private variable containing the name of ...".

1 Messages

The sole purpose of message transmission is to convey information on the state of the sender to the receiver of a message. Therefore, it is necessary to specify explicitly, for every kind of message introduced, what information the message conveys: this helps in reasoning about the protocol.

In the protocol to be presented two types of messages will be used, called "A-messages" and "B-messages":

0. An A-message is a broadcast message and it is empty. It conveys the information that its sender is susceptible, and nothing else.

1. A B-message is a direct message and it is empty too. It conveys the information that its sender is susceptible and that it has the name of the receiver.

Although I have said that these messages are empty, this is not entirely true: they contain their type. So, every message carries one bit of information to distinguish A-messages from B-messages. (In addition, as stated earlier, every message carries the name of its sender.) The distinction between these message types is necessary because they do not convey the same information. Also, I have (tacitly) assumed that the receiver of a message cannot distinguish broadcast messages from direct ones.

remarks: Obvious as it may seem that the information conveyed by messages must be specified explicitly, I have the impression that this principle tends to be violated quite often.

Notice the strict difference in my use of the words "carry" and "convey": even an empty message, carrying no data, conveys information.
□

2 Event handlers

As messages arrive asynchronously, dedicated co-processes are needed to "absorb" arriving messages, so as not to complicate (the code of) the process proper. For this purpose I adopt the following notation to bind a program fragment to a specific condition. This condition just is a guard and its becoming `true` represents an event – that is, a state transition –, like the arrival of a message. Not surprisingly, the program thus bound to the event is sometimes called an "event handler"; this can viewed as a more abstract version of the notion of an "interrupt-service routine".

For boolean expression B and statement S, the construct:

```
on B → S no ,
```

is equivalent to the non-terminating repetition:

```
do true →  await B
           ; { B }
             S
od
```

3 A solution

Every process has a private variable *LocalGroup*, whose value is a set of process names. Variable *LocalGroup* will contain names of susceptible processes only. Initially, *LocalGroup* is empty.

invariant in process p: $(\forall r :: r \in LocalGroup \Rightarrow r$ is susceptible$)$

process p:

```
  { initial state ∧ LocalGroup = φ }
  initialize
; { p is susceptible ∧ LocalGroup = φ }
  add p to LocalGroup
; { p is susceptible ∧ p ∈ LocalGroup }
  send an A-message
```

event handlers for p:

```
on arrival of an A-message from q ∧ p is susceptible
→   { p and q are susceptible }
    add q to LocalGroup
  ; { p and q are susceptible ∧ q ∈ LocalGroup }
    send a B-message to q
no  ,

on arrival of a B-message from q
→   { p and q are susceptible ∧ p ∈ q·LocalGroup }
    add q to LocalGroup
  ; { p and q are susceptible ∧ q ∈ LocalGroup ∧ p ∈ q·LocalGroup }
no  .
```

It is possible to prove the following: if two processes become susceptible and send A-messages, then at least one of them receives an A-message from the other. The subsequent sending of a B-message serves to inform the process that may or may not have received the preceding A-message about the sender's susceptibility.

Also, it is possible to prove that within a finite number of steps the name of every susceptible process is in *LocalGroup* in all susceptible processes.

Remains the problem of *termination detection*: when can a process *decide* that it will receive no more A- or B-messages? If the size of the collection is known in advance to all processes, this is easy: the initialization protocol has terminated if the sizes of all variables *LocalGroup* are equal to the size of the whole collection. But what if this size is not known in advance?

Eindhoven, 20 april 2011

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven