Cancellable Servers: a Pattern for Curiosity

Peter Welch (phw@kent.ac.uk)

CPA 2012 (University of Abertay, 26th. August, 2012)
Curiosity on Mars

This is a student exercise to design and implement part of the control logic for an autonomous robot control process for a rover vehicle on Mars.
This is a student exercise to design and implement part of the control logic for an autonomous robot control process for a rover vehicle on Mars.

The controller has to respond to commands from its operator back on Earth, to operate simple actuators (start/stop motors, deploy gadgets) and to monitor and respond appropriately to input from peripherals (motor feedback clicks, raw echo sensor data, processed camera images).
This is a student exercise to design and implement part of the control logic for an autonomous robot control process for a rover vehicle on Mars.

The controller has to respond to commands from its operator back on Earth, to operate simple actuators (start/stop motors, deploy gadgets) and to monitor and respond appropriately to input from peripherals (motor feedback clicks, raw echo sensor data, processed camera images).

This could be implemented by a purely sequential process … but that’s hard.

A concurrent implementation is simpler, with a process for each external agent (mission control, motor, echo sounder and camera). These processes are linked and communicate as a client/server network.

One twist is that three of the servers must be cancellable (since two transactions need to run in parallel, with the first to complete causing the cancelation of the other).
Curiosity on Mars
Curiosity on Mars

robot.control

monitor. operator

motor

camera

[] hazard

gripper

motor.cmd

log

opr.resp

opr.req
Curiosity on Mars

- opr.req
- opr.resp
- monitor. operator
- monitor. motors
- robot.control
- log
- gripper
- motor.cmd
- hazard
- camera
- motor
Curiosity on Mars

- opr.req
- opr.resp

- monitor. operator
- monitor. motors
- monitor. hazards
- monitor. camera

- log
- gripper
- motor.cmd

- INT
- [] INT
- BOOL
- INT :: [] INT
- BLOB

- motor
- hazard
- camera
Curiosity on Mars

Client

monitor. operator

Server

monitor. motors

distance requested

distance achieved

motor

INT

INT
e.g. [5,5,10,10] for moving forwards, [10,10,5,5] backwards.

array of acceptable hazard levels

Client

monitor. operator

[ ] INT

BOOL

TRUE

(if unacceptable hazard located)

Server

monitor. hazards

[] hazard
Curiosity on Mars

Client

monitor. operator

INT : : [] INT

BLOB

if one is found in one of the given colours

(counted) array of colours

Server

monitor. camera

camera
Curiosity on Mars

robot.control

monitor. operator

monitor. hazards

monitor. motors

BLOB

INT

BOOL

[] INT

INT : : [ ] INT

cancellable servers

motor

[] hazard

camera

log

gripper

motor.cmd

opr.req

opr.resp
Curiosity on Mars

**Normal** client-server transaction:
- the client sends a request, then waits for an answer.
Curiosity on Mars

**Client**

- monitor. operator

**Server**

- monitor. motors

---

**Cancelable** client-server transaction:

- the client sends a request, then waits for an answer;
- while waiting for an answer, the client may give up and cancel the request.

**Problem:**

- if the client tries to cancel and the server tries to answer, then deadlock!
Apart from opening requests, *communications happen in pairs (in parallel, when taking the initiative)*. Now, if both sides take initiative (*client cancel & server answer*), there is no problem! 😊😊😊

**Solution to cancellable** client-server transaction:

- the client sends a *request*, then waits for an *answer*;
  - If an *answer* is received, client sends an *ack* (confirming receipt);
  - to cancel, the client sends a *cancel in parallel* with listening for an *ack*;
  - if an *ack* is received, the request has been cancelled;
  - if an *answer* is received, ignore (server will have seen the *cancel*);
- to answer, the server sends its *answer in parallel* with listening for an *ack*;
  - if an *ack* is received, server knows client accepted the *answer*;
  - if a *cancel* is received, server knows the service was cancelled.
To move robot, first check for hazards (normal client-server transaction). If not clear, don’t move. Otherwise, start motors to move robot …

… then request **monitor.motors** to say when enough clicks have been seen **and** request **monitor.hazards** to look out for specified hazards. Listen to both servers for answers. Whoever answers first, cancel the other!
To find a blob, start motors to turn robot …

… then request `monitor.motors` to say when enough turn has been done and request `monitor.camera` to look out for specified colour(s). Then, …

… listen to both servers for answers. Whoever answers first, cancel the other!
Implement and Verify ...
Curiosity on Mars

```
PROTOCOL ASK
  CASE
    ask, INT
    cancel
    ans.ack
    :

PROTOCOL ANS
  CASE
    ans, INT
    cancel.ack, INT
    -- target best effort
```

-- target sought
-- target found

watcher
**Curiosity on Mars**

**PROC** `simple.watcher` (CHAN `ASK` `in?`, CHAN `ANS` `out!`, CHAN `INT` `data?`)

```
WHILE TRUE
  PRI ALT
    INT `target`:
    in ? ask; target              -- service requested
    INT `d`:
    SEQ
      data ? `d`
      WHILE `d` <> `target`
        data ? `d`
        out ! ans; target
      INT `d`:
      data ? `d`
      SKIP
      :  
```

---

 >::
Curiosity on Mars

But *simple.watcher* does not deal with a *cancel* request ...

First, let’s try it the obvious, but wrong, way ...
PROC bad watcher (CHAN ASK in?, CHAN ANS out!, CHAN INT data?)
WHILE TRUE
  PRI ALT
  INT target:
  in ? ask; target                                -- service requested
  INITIAL BOOL serving IS TRUE:
  WHILE serving
    PRI ALT
    in ? cancel                      -- service cancelled
      serving := FALSE
    INT d:
    data ? d                          -- monitor and check
    IF
      d = target
      SEQ
      out ! ans; target              -- service result
      serving := FALSE
    TRUE
      SKIP
    SKIP
  INT d:
  data ? d                                -- monitor and discard
  SKIP
  SKIP
::
PROC bad.control (VAL INT timeout, CHAN ASK in?, CHAN ANS out!,
CHAN ASK to.server!, CHAN ANS from.server?)

WHILE TRUE
  TIMER tim:
  INT t, target:
  SEQ
    in ? ask; target               -- from mission control
    to.server ! ask; target        -- request service
    tim ? t
  ALT
    from.server? ans; target       -- service result
    out ! ans; target               -- to mission control
    tim ? AFTER t PLUS timeout
      SEQ
        to.server ! cancel          -- (or PAR)
        out ! ans; -1              -- cancel service
        out ! ans; -1              -- to mission control
  :

PROC bad.system (VAL INT timeout, CHAN ASK in?, CHAN ANS out!, CHAN INT data?)
CHAN ASK to.server:
CHAN ANS from.server:
PAR
bad.control (timeout, in?, out!, to.server!, from.server?)
bad.watcher (to.server?, from.server!, data?)

But …

VERIFY LIVELOCK.FREE bad.system -- reassuring
VERIFY NOT DEADLOCK.FREE bad.system -- expected, 😊
Curiosity on Mars

Let’s do it right …
PROC cancellable.watcher (CHAN ASK in?, CHAN ANS out!,
CHAN INT data?)

WHILE TRUE
  PRI ALT
    INT target:
    in ? ask; target -- service requested
    INITIAL BOOL serving IS TRUE:
    WHILE serving
      PRI ALT
      ... deal with service cancellation
      ... deal with sensor data
    INT d:
    data ? d -- monitor and discard
    SKIP
  :

{{
  deal with service cancellation
  in ? cancel -- service cancelled
  SEQ
  out ! cancel.ack; target -- acknowledge cancel
  serving := FALSE
}}
{{{ deal with sensor data

INT d:
data ? d                      -- monitor and check

  IF
d = target
  SEQ

    PAR
      out ! ans; target  -- service result
      in ? CASE
      ans.ack            -- result accepted
      SKIP
      cancel             -- result 'ignored'
      SKIP
      serving := FALSE
      TRUE
      SKIP

    SKIP

}}}
PROC bad.control (VAL INT timeout, CHAN ASK in?, CHAN ANS out!, CHAN ASK to.server!, CHAN ANS from.server?)

WHILE TRUE
    TIMER tim:
    INT t, target:
    SEQ
        in ? ask; target
        to.server ! ask; target
        tim ? t
    ALT
        from.server? ans; target
        out ! ans; target
        -- service result
        -- to mission control
        tim ? AFTER t PLUS timeout
        SEQ
            to.server ! cancel
            out ! ans; -1
            -- (or PAR)
            -- cancel service
            -- to mission control

: 
PROC good.control (VAL INT timeout, CHAN ASK in?, CHAN ANS out!,
CHAN ASK to.server!, CHAN ANS from.server?)

WHILE TRUE
  TIMER tim:
    INT t, target:
    SEQ
      in ? ask; target               -- from mission control
      to.server ! ask; target        -- forward request
      tim ? t
    ALT
      from.server? ans; target      -- service result
        SEQ
          to.server ! ans.ack          -- (or PAR)
          out! ans; target            -- acknowledge result
      tim ? AFTER t PLUS timeout    -- (cannot be PAR)
        SEQ
          PAR
            to.server ! cancel         -- cancel service
            from.server ? CASE
              ans; target              -- accept as acknowledge
              SKIP
            cancel.ack; target        -- actual acknowledge
              SKIP
          out! ans; target             -- to mission control
  :
PROC good.system (VAL INT timeout, CHAN ASK in?, CHAN ANS out!, CHAN INT data?)

CHAN ASK to.server:
CHAN ANS from.server:
PAR
    good.control (timeout, in?, out!, to.server!, from.server?)
    cancellable.watcher (to.server?, from.server!, data?)

And ...

VERIFY LIVELOCK.FREE good.system -- reassuring
VERIFY DEADLOCK.FREE good.system -- gotcha 😊😊😊
Curiosity on Mars

The exercise is to design and implement an autonomous `robot.control` process for a rover vehicle on Mars.

The controller has to respond to commands from its operator back on Earth, to operate simple actuators (start/stop motors, deploy gadgets) and to monitor and respond appropriately to input from peripherals (motor feedback clicks, raw echo sensor data, processed camera images).

The controller must not deadlock … or have a sub-system deadlock …

For Curiosity (or any autonomous vehicle), the verification is not yet sufficient … we need to verify that multiple cancellable servers do not cause problems …
To find a blob, start motors to turn robot …

… then request **monitor.motors** to say when enough turn has been done and request **monitor.camera** to look out for specified colour(s). Then, …

… listen to both servers for answers. Whoever answers first, cancel the other!
Curiosity on Mars

To find a blob, start motors to turn robot …

… then request `monitor.motors` to say when enough turn has been done and request `monitor.camera` to look out for specified colour(s). Then, …

… listen to both servers for answers. Whoever answers first, cancel the other!

Just changing the names …
Curiosity on Mars

VERIFY LIVELOCK.FREE curiosity -- reassuring

VERIFY DEADLOCK.FREE curiosity -- not enough !!!
The problem is ...

```
VERIFY LIVELOCK.FREE bad.curiosity  -- reassuring
VERIFY DEADLOCK.FREE bad.curiosity  -- surprise !!! 😎😎😎
```
The problem is ...

The **green sub-system** may deadlock, leaving **bad.watcher** still alive (in its outer loop) accepting and discarding camera data forever. So, the system is not deadlocked!
The problem is ...

Or the pink sub-system may deadlock, leaving bad.counter still alive (in its outer loop) accepting and discarding camera data forever. So, the system is not deadlocked!
Consider:

```
Consider:

bad.robot.
control

bad. counter

bad. watcher

bad.curiosity

motor

camera

opr.req

opr.resp
```
Consider:

**Curiosity on Mars**

- **bad.curiosity.sensors**
  - bad.robot.control
  - bad.counter
  - bad.watcher

- **black.hole.sensors**
  - Motor
  - Camera
Curiosity on Mars

Q.E.D.

- bad.curiosity.sensors
- bad.robot.control
- bad.counter
- bad.watcher

motor

camera

VERIFY LIVELOCK.FREE bad.curiosity.sensors -- reassuring
VERIFY LIVELOCK.FREE black.hole.sensors -- reassuring

VERIFY NOT bad.curiosity.sensors REFINES.F black.hole.sensors -- gotcha !!!
Curiosity on Mars

Q.E.D.

Verify LIVELOCK.FREE good.curiosity.sensors -- reassuring
Verify LIVELOCK.FREE black.hole.sensors -- reassuring
Verify curiosity.sensors REFINES.F black.hole.sensors -- gotcha !!!
Now, **black.hole.sensors** never refuses motor or camera. Therefore, neither does **curiosity.sensors** (nor **curiosity**).
Curiosity on Mars

Q.E.D.

Now, \texttt{black.hole.sensors} never refuses motor or camera. Therefore, \texttt{bad.curiosity.sensors} (and \texttt{bad.curiosity}) does.
Curiosity on Mars

This is a student exercise to design and implement part of the control logic for an autonomous robot. The controller has to respond to commands from its operator back on Earth, to operate simple actuators (start/stop motors, deploy gadgets) and to monitor and respond appropriately to input from peripherals (motor feedback clicks, raw echo sensor data, processed camera images).

The controller must not deadlock … or have a sub-system deadlock …

Run Other Demo …
Curiosity on Mars

- keyboard
- drive
- curiosity
- cancellable servers
- motor processor
- camera processor
- reporter
- screen

Other Demo …
Curiosity on Mars

- Drive
- bad.creativity
- motor. processor
- camera. processor
- bad.cancellable. servers
- reporter
- screen
Curiosity on Mars

- drive
- simple.curiosity
- motor. processor
- camera. processor
- simple. servers
- reporter
- screen
- keyboard

Other Demo …
Curiosity on Mars

Source codes for the system in this presentation is available in 3 forms:

- cancellable-servers.op2
- cancellable-servers.csp
- cancellable-servers.occ

**cancellable-servers.csp**

**CSP** script (showing **occam-π²** source code) – FDR ready.

Executable **occam-π** source code (with **VERIFY** assertions/**PROC**s commented out) – includes testrig.

* For now, generated by hand …
Curiosity on Mars

This is a student exercise to design and implement part of the control logic for an autonomous robot process for a rover vehicle on Mars.

Any questions?