A Process-Oriented Architecture for Complex System Modelling

Peter Welch and Carl Ritson
Computing Laboratory, University of Kent at Canterbury

IFIP WG 2.4 (UCLA, Lake Arrowhead, 7th. May, 2007)

Case Study: blood clotting

Haemostasis: we consider a greatly simplified model of the formation of blood clots in response to damage in blood vessels.

Platelets are passive quasi-cells carried in the bloodstream. They become activated when a balance between chemical suppressants and activators shift in favour of activation.

When activated, they become sticky ...

We are just going to model the clumping together of such sticky activated platelets to form clots.

To learn and refine our modelling techniques ...

Case Study: blood clotting

Case Study: blood clotting

Case Study: blood clotting

Case Study: blood clotting

Rapid Review: CSP

CSP deals with processes, networks of processes and various forms of synchronisation / communication between processes.

A network of processes is also a process - so CSP naturally accommodates layered network structures (networks of networks).

We do not need to be mathematically sophisticated to work with CSP. That sophistication is pre-engineered into the model. We benefit from this simply by using it.
Processes

- A process is a component that encapsulates some data structures and algorithms for manipulating that data.
- Both its data and algorithms are private. The outside world can neither see that data nor execute those algorithms! [They are not objects …] [Think chips …]
- The algorithms are executed by the process in its own thread (or threads) of control.
- So, how does one process interact with another?

Synchronised Communication

A may write on c at any time, but has to wait for a read.

B may read from c at any time, but has to wait for a write.

Synchronised Communication

Only when both A and B are ready can the communication proceed over the channel c.

Simple Deadlock Example

If there is no discipline on when A and B communicate, then A may commit to output on c, followed by B on d … or vice-versa. Either way, neither are listening and both are stuck. Same happens if both commit to input.

Careful
- Writer blocks if reader is not ready
-Lots of deadlock possibilities

Benefit

- Once the writer has written, it knows the reader has read

OK: plenty of other processes to run and ultra-fast context switch (comparable to a procedure call)

OK: work with a small set of synchronisation patterns for which we have proven safety theorems
Client-Server Pattern

- **Client**: makes a request any time, then commits to taking reply.
- **Server**: always accepts a request (within some bounded time), then always makes a reply (within some bounded time). It may make requests itself, as a client to other servers.

No deadlock is now possible from this client-server relationship.

Client-Server Pattern

- **Client**: makes a request any time, then commits to taking reply.
- **Server**: always accepts a request (within some bounded time), then always makes a reply (within some bounded time). It may make requests itself, as a client to other servers.

Symbology: this represents a client-server relation. It points to the server and allows a two-way conversation (initiated by the client).

Client-Server Pattern

A server may have many clients …

Only one client at a time converses with the server. They form an orderly queue. Still no deadlock possible – and no starvation. No polling on the queue, so no livelock either.

Client-Server Pattern

A client-server system that has no cycles in its client-server relationships is deadlock, livelock and starvation free.

Barrier Synchronisation

The **occam-π BARRIER** type corresponds to a multiway CSP event, though some higher level design patterns (such as resignation) have been built in.

Basic CSP semantics apply. When a process synchronises on a barrier, it blocks until all other processes enrolled on the barrier have also synchronised. Once the barrier has completed (i.e. all enrolled processes have synchronised), all blocked processes are rescheduled for execution.

Barrier Synchronisation

The **occam-π BARRIER** type corresponds to a multiway CSP event, though some higher level design patterns (such as resignation) have been built in.

A PAR construct must explicitly ENROLL its components on barriers.
Barrier Synchronisation

Processes may synchronise on more than one barrier:

```
worker (0)  worker (1)  ...  worker (n-1)
```

To synchronise on a barrier:

```
SYNC b  or  SYNC c
```

Barrier Synchronisation

Barriers are commonly used to synchronise multiple phases of computation between a set of processes. Within each phase, other synchronisations (channel/barrier) may take place:

```
PROC worker (VAL INT id, BARRIER b, c)
    ...  local declarations / initialisation
    WHILE running
        SEQ
        SYNC b
        ...  phase b computation
        SYNC c
        ...  phase c computation
```

Of course, only one barrier is actually needed to synchronise the phases in this example:

```
PROC worker (VAL INT id, BARRIER b)
    ...  local declarations / initialisation
    WHILE running
        SEQ
        SYNC b
        ...  phase 0 computation
        SYNC b
        ...  phase 1 computation
```

Try to break this rule … your program won’t compile. There are zero memory and run-time costs to enforce it.

Deferred Client-Server Pattern

Sometimes, a server cannot reply to a client until it has been given information from all its clients. [This is like a barrier synchronisation, but with a message exchange.]

If we know the server and all its clients are in that state, it is safe to proceed …

Deferred Client-Server Pattern

Ocaml barrier synchronisation is safe in the sense that enrollment and resignation are automatically managed. A process may synchronise on a BARRIER if and only if it is enrolled.

Try to break this rule … your program won’t compile. There are zero memory and run-time costs to enforce it.

Deferred Client-Server Pattern

If the server knows how many clients (say n) will be calling …

... each client calls the server twice: first on a channel to give its information and, then, on a different channel to request the server’s answer. The server takes n pieces of information on its first service channel, then delivers its answer n times down the other. [Note: each transaction still conforms to the basic client-server pattern.]
**Deferred Client-Server Pattern 2**

If the server does not know how many clients will be calling ...

... each client calls the server twice: first to give its information and, then, to request the server’s answer. In between, they barrier synchronise with each other. The server first consumes information on its service channel; when an answer request arrives, it knows all the clients have reported, computes its answer and delivers it as many times as asked. [Note: each transaction still conforms to the basic client-server pattern.]

---

**An Architecture for Emergence**

We propose a layered dynamic client-server phased-barrier architecture.

**Bottom Layer**

'Site' processes: pure servers representing a locality. They hold state information defining world topology (by holding client-ends of the service channels of neighbouring sites) and local environmental data (such as the presence of mobile agents, chemical factors, permeability, flow vectors). The topology is usually a Euclidean grid and fixed, though this is not essential.

**Second Layer**

Mobile 'agent' processes: pure clients (of site servers and higher-level controllers). They are attached to sites through which they observe the world and change it. They move from site to site under their own (and their controller’s) rules. They keep in step through barrier synchronisation, following a simple cycle ...
An Architecture for Emergence

To swing down a chain of 1M sites, exchanging one INT during each visit: 770 nsecs/visit (P3), 280 nsecs/visit (P4)

To swing down a chain of 1M sites, but doing no business: 450 nsecs/visit (P3), 120 nsecs/visit (P4)

Location (Neighbourhood) Awareness

The Matrix

Mobile Agents

Location (Neighbourhood) Awareness

Location (Neighbourhood) Awareness
**Dynamic Construction of Agents**

Forking

**Blood Clotting Model**

**Site Processes (bottom layer)**

Pure servers. Define a 3D Euclidean grid. Each site has 26 neighbours. They allow one platelet at a time — any number of chemical factors.

Stores blood vessel wall flag (no agents may enter if set), a voxel (from a 3D array spanning all sites, coloured if platelet is present), blood flow vector.

If a platelet is present, it holds the client-end of the platelet's clotting controller process (two layers up). It may give this to an enquiry from a platelet on a neighbouring site.

**Platelet Processes (second layer)**

Pure clients. Drive the whole model. Engage on discovery and modify barriers. Communicate with and move between site servers and their clotting controllers. Smooth or sticky (initially smooth).

**Chemical Factor Processes (second layer)**

Initial vector away from wound. Modifies factor level and flow vector in sites. Eventually drawn into blood flow.

**Clot Processes (third layer)**

Controlling servers for groups of stuck-together platelets.

Accumulate flow vectors from platelets and information on adjacent sites (e.g. bumped into another clot?).

If sufficiently hard bump (possibly into more than one clot), engages in election with bumped clots to see who takes over all the platelets in a merged clot. Terminates if loses.

Otherwise, informs combined set of platelets of their new controller.

**Clot Processes (third layer)**

Deferred client-server pattern 1
**Blood Clotting Model**

*Platelet Processes (second layer) – again ...*

**Discovery phase.** Get chemical factors, flow vector and clot information in the neighbourhood from site. If smooth and enough factor, fork off clot controller and become sticky. If sticky, forward flow vector and neighbourhood info to clot for movement decision – else make it ourself. Receive info from clot on possible new clot controller and movement decision. *If moving, offer move to target site.*

**Modify phase.** If not moving, this is null. *If trying to move, get decision from target site* – other clots may have tried to move there. If OK, move.

---

**3-D Bloodstream**

---

**3-D Bloodstream**

---

**3-D Bloodstream**

---

**3-D Bloodstream**
3-D Bloodstream

3-D Bloodstream

3-D Bloodstream

3-D Bloodstream
3-D Bloodstream

3-D Bloodstream

3-D Bloodstream

3-D Bloodstream

3-D Bloodstream

3-D Bloodstream
The biological accuracy of the current model is very approximate. Nevertheless …

**Emergent Behaviour**

Clots do form in apparently realistic ways. They gather round the wound, form a plug and staunch the loss of blood (deduced from the flow of platelets – we are not directly modeling blood).

Simple experiments are possible – for example, varying the concentration of platelets. Too high and the clots form too fast around the leading edge of the wound, block platelet traffic and fail to seal the hole. Too low and the clots form too slow and get swept away. Nature demands a balance!

The accuracy can be refined in a stable way to greater and greater realism. More agent types can be added – clot busting factors, proteins (like fibrin that make up the bulk of real clots). Existing agent types can be refined. Introduce time lag delaying the onset of a sufficiently stimulated platelet (so it can get a little way past initial encounters with existing clots).
**Conclusions**

The biological accuracy of the current model is very approximate. Nevertheless …

**Emergent Behaviour**

Clots do form in apparently realistic ways. They gather round, form a plug and staunch the loss of blood (deduced from the flow of platelets – we are not directly modeling blood).

Simple experiments are possible – for example, varying the concentration of platelets. Too high and the clots form too fast around the leading edge of the wound, block platelet traffic and fail to seal the hole. Too low and the clots form too slow and get swept away. Nature demands a balance!

**Acknowledgements**

The work reported here was supported by the (UK EPSRC funded) **TUNA** project. Partners in this project were the Computer Science departments of the Universities of York, Kent and Surrey:

http://www.cs.york.ac.uk/nature/tuna/

**Appendix:**

**Story Boards**

The following slides show a **story-board** sequence of process network diagrams. They illustrate an evolving network showing the movement of clots through a 1-D (for simplicity here) blood vessel, including a merger of two bumping clots into one. The system presented was 3-D.

Process movement is implemented by **mobile channel-ends**.

Elements of the supporting process network (platelet generation, user interaction and display) are also shown.

Such storyboards (similar to those used for making movies) are a key element of dynamic process-oriented design. The program code for the clot process was directly written from these diagrams … **and worked correctly first time.**
Platelet Model (1-D)

Phase 0

Phase 1

Platelet Model (1-D)