School of Computing Science

60 YEARS OF COMPUTING AT GLASGOW

Message Passing (2)

Advanced Operating Systems

Lecture 12

Lecture Outline

- Use of message passing
  - Pattern matching and state machines
  - Remote actors
  - System upgrade and evolution
- Error handling in message passing systems

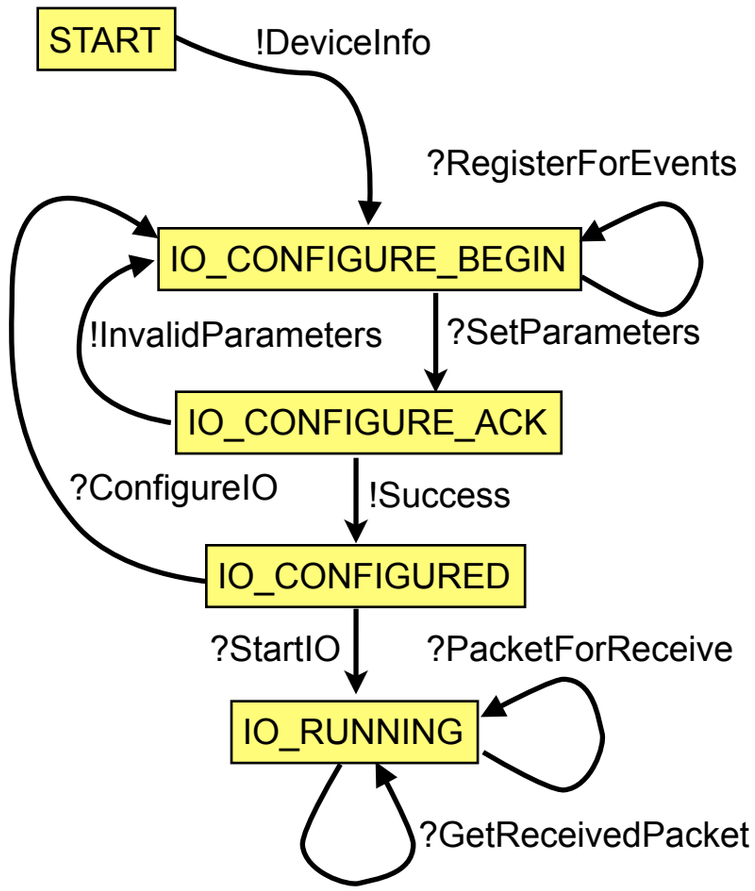
Patterns and State Machines

- A set of states and transitions triggered by/causing events forms a state machine
  - An actor comprises a set of events – *messages* – and various states – *functions* – that process events as they are received
  - Pattern matching operation dictates response to different types of events in each state

- Discussed the idea for device driver robustness – but natural for message passing actors
  - Message passing code naturally contains a formalised description of the state machine

Example: Singularity State Machines

- Singularity devices drivers are an example formal state machine in a message passing system




```
contract NicDevice {
  out message DeviceInfo(...);
  in  message RegisterForEvents(NicEvents.Exp:READY
  c);
  in  message SetParameters(...);
  out message InvalidParameters(...);
  out message Success();
  in  message StartIO();
  in  message ConfigureIO();
  in  message PacketForReceive(byte[] in ExHeap p);
  out message BadPacketSize(byte[] in ExHeap p, int
  m);
  in  message GetReceivedPacket();
  out message ReceivedPacket(Packet * in ExHeap p);
  out message NoPacket();

  state START: one {
    DeviceInfo! → IO_CONFIGURE_BEGIN;
  }
  state IO_CONFIGURE_BEGIN: one {
    RegisterForEvents? →
      SetParameters? → IO_CONFIGURE_ACK;
  }
  state IO_CONFIGURE_ACK: one {
    InvalidParameters! → IO_CONFIGURE_BEGIN;
    Success! → IO_CONFIGURED;
  }
  state IO_CONFIGURED: one {
    StartIO? → IO_RUNNING;
    ConfigureIO? → IO_CONFIGURE_BEGIN;
  }
  state IO_RUNNING: one {
    PacketForReceive? → (Success! or BadPacketSize!)
      → IO_RUNNING;
    GetReceivedPacket? → (ReceivedPacket! or
    NoPacket!)
      → IO_RUNNING;
    ...
  }
}
```

Listing 1. Contract to access a network device driver.

[G. Hunt and J. Larus. Singularity: Rethinking the software stack. ACM SIGOPS OS Review, 41(2), Apr. 2007. DOI 10.1145/1243418.1243424]

Example: Singularity State Machines

- Contract defines the state machine – essentially an abstract type
- Implementation uses pattern matching against received messages
  - A function for each state
  - Each function switches based on type of the message object received

```
NicDevice.Exp:IO_RUNNING nicClient ...

switch receive {
  case nicClient.PacketForReceive(buf):
    // add buf to the available buffers, reply
    ...

  case nicClient.GetReceivedPacket():
    // send back a buffer with packet data if available
    ...

  case nicClient.ChannelClosed():
    // client closed channel
    ...
}
```

the state

messages that can be received in that state

[M. Fähndrich *et al.* Language support for fast and reliable message-based communication in Singularity OS. Proc. EuroSys 2006. DOI 10.1145/1218063.1217953]

- Compiler checks `switch receive` statements handle all messages defined by the contract
  - Blocks in the switch receive statement must end with a transfer of control, to a function representing a new state or to itself, allowing compiler to check transitions

Modelling State Machine Correctness

- If state machine is formally defined in code, can begin to verify it
  - Check that the code implements the defined state machine
  - Check the state machine itself
    - Validate that the driver cannot deadlock
    - Validate that certain states can be reached
    - …
    - [discussed further in the MRS4 course]

- Code can readily be translated into (fragments of) a Promela model, for example, suitable for verification with a model checker such as SPIN

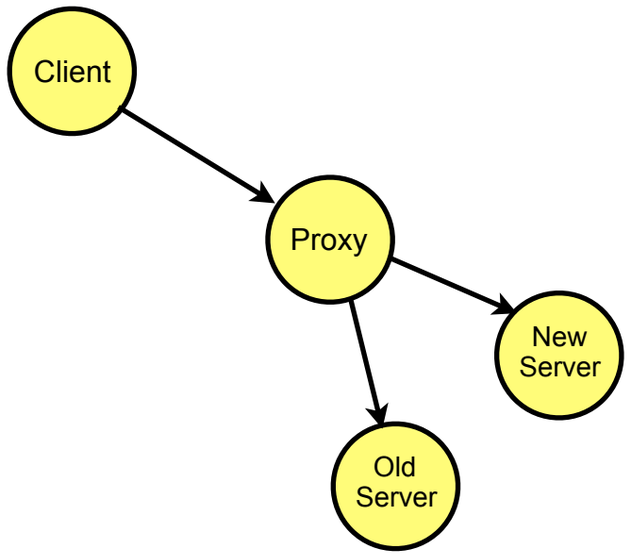
Remote Actors

- Two approaches to identifying message receiver:
  - Receiver is anonymous, but bound to named channel
  - Receiver is explicitly named as message destination
- Both required a *named* destination for messages
  - Trivial to make this an opaque URL for the application, but meaningful to the runtime – can identify remote actors
  - Since messages either immutable or linearly typed, data can be safely copied across the network

- Most message passing systems allow transparent use of remote actors

System Upgrade and Evolution

- Message passing allows for easy system upgrade
  - Rather than passing messages directly to server, pass via proxy
  - Proxy can load a new version of the server and redirect messages, without disrupting existing clients
  - Eventually, all clients are talking to the new server; old server is garbage collected
- Allows for gradual transparent system upgrade
  - A running system can be upgraded without disrupting service



- Use of dynamic typing can make the upgrade easier
  - New components of the system can generate additional messages, which are ignored by old components
  - Supervisor hierarchy allows system to notice if components fail, and fallback to known good version
  - Backwards compatible extensions are simple to add in this manner

Error Handling

- The system is massively concurrent – errors in one part can be handled elsewhere
- Error handling philosophy in Erlang:
  - Let some other process do the error recovery
  - If you can't do what you want to do, die
  - Let it crash
  - Do not program defensively

J. Armstrong, "Making reliable distributed systems in the presence of software errors", PhD thesis, KTH, Stockholm, December 2003, http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf

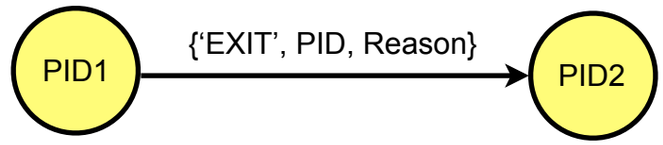
- Be concerned with the overall system reliability, not the reliability of any one component

Let It Crash

- In a single-process system, that process must be responsible for handling errors
  - If the single process fails, then the entire application has failed
- In a multi-process system, each individual process is less precious – it's just one of many
  - Changes the philosophy of error handling
  - A process which encounters a problem should not try to handle that problem – instead, fail loudly, cleanly, and quickly "let it crash"
  - Let another process cleanup and deal with the problem
  - Processes become much simpler, since they're not cluttered with error handling code

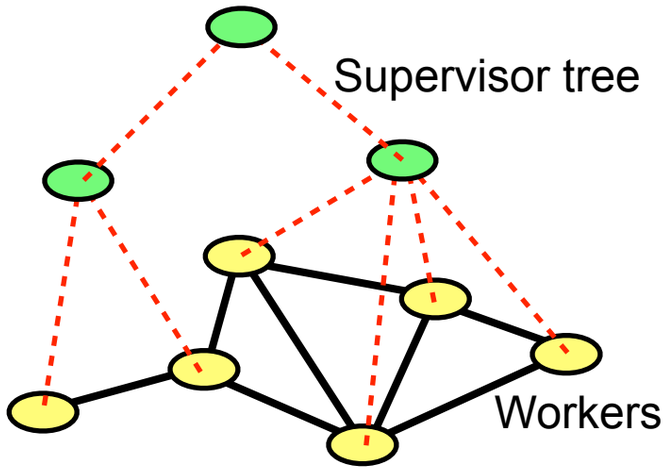
Remote Error Handling

- How to handle errors in a concurrent distributed system?
  - Isolate the problem, let an unaffected process be responsible for recovery
  - Don’t trust the faulty component
  - Analogy to hardware fault tolerance
- Processes are linked, and the runtime is set to trap errors and send a message to the linked process on failure
  - e.g., process PID2 has requested notification of failure of PID1; runtime sends an “EXIT” message on failure, to tell PID2 that PID1 failed, and why
  - Process PID2 then restarts PID1, and any other dependent processes



Supervision Hierarchies

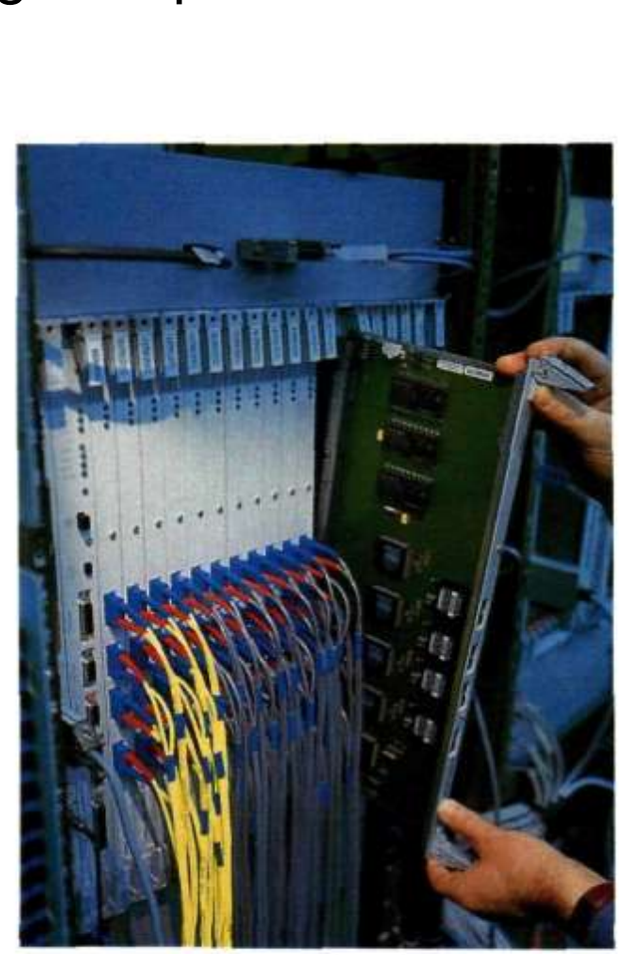
- Organise problems into tree-structured groups of processes, letting the higher nodes in the tree monitor and correct errors in the lower nodes
  - Supervision trees are trees of supervisors – processes that monitor other processes in the system
  - Supervisors monitor workers – which perform tasks – or other supervisors
  - Workers are instances of behaviours – processes whose operation is characterised by callback functions (i.e., the Erlang equivalent of objects)
    - E.g., server, event handler, finite state machine, supervisor, application



- Abstract common behaviours into objects
- Workers managed by supervisor processes that restart them in the case of failure, or otherwise handle errors

Robustness of Erlang Systems

- Example: Ericsson AXD301 ATM switch
  - Dimensioned to handle ~50,000 simultaneous flows with ~120 in setup or teardown phase at any one time
  - Processes ATM traffic at 160 gigabits per second (16 x 10Gbps links)
  - ~1.1 million lines of Erlang in 2248 Erlang modules
  - ~40 programmers



Images from: S. Blau, J. Rooth, J. Axell, F. Hellstrand, M. Buhrgard, T. Westin, and G. Wicklund, “AXD 301: A new generation ATM switching system”, Ericsson Review, 1998.

Robustness of Erlang Systems

- Example: Ericsson AXD301 ATM switch
  - 99.9999999% reliable in real-world deployment on 11 routers at a major Ericsson customer (~0.5 seconds downtime per year)
  - Yet, failures do occur, and are handled by the supervision hierarchy and distributed error recovery
  - Employs restart-and-recover semantics per-connection
  - Failures may disrupts one connection out of tens-of-thousands – assumes failures are transient; system doesn't employ multi-version programming

Discussion

- The let-it-crash philosophy changes error handling, moving it out-of-process
- There are a few compelling case studies to show it can work well in some domains
- Is this a generally appropriate error-handling tool?

Further Reading

- J. Armstrong, “Erlang”, Communications of the ACM, 53(9), September 2010, DOI:10.1145/1810891.1810910

- Does the programming model make sense?
- Does the reliability model (“let it crash”) make sense?

contributed articles

DOI:10.1145/1810891.1810910

The same component isolation that made it effective for large distributed telecom systems makes it effective for multicore CPUs and networked applications.

BY JOE ARMSTRONG

Erlang

ERLANG IS A concurrent programming language designed for programming fault-tolerant distributed systems at Ericsson and has been (since 2000) freely available subject to an open-source license. More recently, we've seen renewed interest in Erlang, as the Erlang way of programming maps naturally to multicore computers. In it the notion of a process is fundamental, with processes created and managed by the Erlang runtime system, not by the underlying operating system. The individual processes, which are programmed in a simple dynamically typed functional programming language, do not share memory and exchange data through message passing, simplifying the programming of multicore computers.

Erlang[2] is used for programming fault-tolerant, distributed, real-time applications. What differentiates it from most other languages is that it's a concurrent programming language; concurrency belongs to the language, not to the operating system. Its programs are collections of parallel processes cooperating to solve a particular problem that can be created quickly and have only limited memory overhead; programmers can create large numbers of Erlang processes yet ignore any preconceived ideas they might have about limiting the number of processes in their solutions.

All Erlang processes are isolated from one another and in principle are “thread safe.”[a] When Erlang applications are deployed on multicore computers, the individual Erlang processes are spread over the cores, and programmers do not have to worry about the details. The isolated processes share no data, and polymorphic messages can be sent between processes. In supporting strong isolation between processes and polymorphism, Erlang could be viewed as extremely object-oriented though without the usual mechanisms associated with traditional OO languages.

Erlang has no mutexes, and processes cannot share memory.[a] Even within a process, data is immutable. The sequential Erlang subset that executes within an individual process is a dynamically typed functional programming language with immutable state.[b] Moreover, instead of classes, methods, and inheritance, Erlang has modules that contain functions, as well as higher-order functions. It also includes processes, sophisticated error handling, code-replacement mechanisms, and a large set of libraries.

Here, I outline the key design criteria behind the language, showing how they are reflected in the language itself, as well as in programming language technology used since 1985.

Shared Nothing

The Erlang story began in mid-1985 when I was a new employee at the Ericsson Computer Science Lab in Stock-

a The shared memory is hidden from the programmer. Practically all application programmers never use primitives that manipulate shared memory; the primitives are intended for writing special system processes and not normally exposed to the programmer.

b This is not strictly true; processes can mutate local data, though such mutation is discouraged and rarely necessary.

68 COMMUNICATIONS OF THE ACM | SEPTEMBER 2010 | VOL. 53 | NO. 9