Message Passing (2)

Advanced Operating Systems
Lecture 12
Lecture Outline

• Message handling
• Pattern matching and state machines
• Remote actors
• System upgrade and evolution
• Error handling in message passing systems
Message Handling

• Receivers pattern match against messages
  • Match against message types, not just values
  • Type system can ensure an exhaustive match

• Messages queued for processing
  • Enqueue operation is non-blocking
  • Dispatcher manages a thread pool servicing receiver components of the actors
  • Receivers operate in message processing loop – single-threaded, with no concern for concurrency
  • Sent messages enqueued for processing by other actors
Use Immutable Messages

- Runtime ensures a receiver processes messages sequentially, but it is part of a concurrent system
  - Sending and receiving actors may run concurrently
  - Message data is shared between sender and receiver

- Important to ensure message data is immutable
  - Or, at least, never mutated once the message has been sent
  - Erlang ensures this in the language → races due to shared message data not possible
  - Scala+Akka requires programmer discipline → potential race conditions if message data modified after message sent
Ownership Transfer – Linear Types

• Alternative to immutability: type system ensures ownership of message data is transferred

• A variable with linear type may be used only once; it goes out of scope after use

• Potentially useful when sharing mutable data between threads
  • Implement sharing via a send function that takes a linear type for the data to be shared
  • Message data consumed by send function and receiver, so can’t be used by the sender after message has been sent
  • Data doesn’t need to be locked, since it can only be used by one thread at once

• The compiler enforces that linear data is not shared between threads

let x = ~5;
let y = ~x;
let z = *x + 1;  // error

let x = 5;
let y = foo(x);
dest.send(y);
let z = y + 1  // error


Rust programming language: http://rust-lang.org/
Efficiency of Message Passing

- Assuming immutable message or linear types, message passing has an efficient implementation
  - Copy message data in distributed systems
  - Pass pointer to data in shared memory systems
  - Neither case needs to consider shared access to message data

- Garbage collected systems often allocate messages from a shared exchange heap
  - Collected separately from per-process heaps
  - Expensive to collect, since data in exchange heap owned by multiple threads – need synchronisation
  - Per-process heaps can be collected independently and concurrently – ensures good performance

[Fig. 2. The Exchange Heap.]

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);
  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.

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 the type of the message object received
- Compiler checks that `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
- Messages are immutable messages sent between actors

```c
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 ...
}
```

Modelling State Machine Correctness

• If state machine is formally defined in code, we 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

• 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
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?