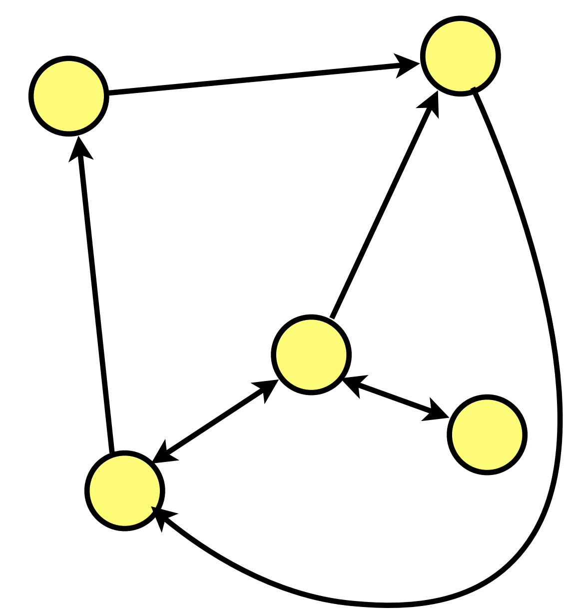
University of Glasgow | School of Computing Science

Message Passing Systems

Advanced Operating Systems (M)
Lecture 19

Message Passing

- System is structured as a set of communicating processes, with no shared mutable state
- All communication via exchange of messages
  - Messages are generally required to be immutable – data is conceptually copied between processes
  - Some systems use linear types to ensure messages are not referenced after they are sent, allowing mutable data to be safely transferred



- Implementation
  - Implementation within a single system usually built with shared memory and locks, passing a reference to the message
  - Trivial to distribute, by sending the message down a network channel – the runtime needs to know about the network, but the application can be unaware that the system is distributed

Interaction Models

- Message passing can involve rendezvous between sender and receiver
  - A *synchronous* message passing model – sender waits for receiver
  - e.g., occam2
- Alternatively, communication may be asynchronous
  - The sender continues immediately after sending a message
  - Message is buffered, for later delivery to the receiver
  - e.g., Erlang, Scala actors, Singularity channels
  - Synchronous rendezvous can be simulated by waiting for a reply

Communication and the Type System

- Statically-typed communication
  - Explicitly define the types of message that can be transferred
  - Compiler checks that receiver can handle all messages it can receive – robustness, since a receiver is guaranteed to understand all messages
  - e.g., Singularity
- Dynamically-typed communication
  - Communication medium conveys any time of message; receiver uses pattern matching on the received message types to determine if it can respond to the messages
  - Potentially leads to run-time errors if a receiver gets a message that it doesn't understand
  - e.g., Erlang, Scala Actors

Naming of Communications

- Are messages sent between named processes or indirectly via channels?
    - Erlang and Scala directly send messages to processes, each of which has its own mailbox
    - Singularity and occam2 require explicit *channels* to be created, with messages being sent indirectly via the channel

    - Explicit channels require more plumbing, but the extra level of indirection between sender and receiver may be useful for evolving systems
    - Explicit channels are a natural place to define a communications protocol for statically typed messages

Erlang and Scala

- Two widely deployed message passing systems:
  - Erlang (http://www.erlang.org/)
  - Scala (http://www.scala-lang.org/)
    - Scala is an open-source multi-paradigm (functional/object-oriented) programming language that runs on the JVM, and seamlessly interoperates with Java code
    - The bundled *actors* library gives Erlang-like concurrency primitives

J. Armstrong, "Erlang", CACM, 53(9), DOI:10.1145/1810891.1810910

- Both adopt a similar message passing model:
  - Asynchronous – messages are buffered at receiver; sender does not wait
  - Dynamically typed – any type of message may be sent to any receiver
  - Messages sent to named processes, not via channels
- Both provide transparent distribution of processes in a networked system

Message Passing: Scala Example

```
class Ping(count: int, pong: Actor) extends Actor {
  def act() {
    var pingsLeft = count - 1
    pong ! Ping
    loop {
      react {
        case Pong =>
          if (pingsLeft % 1000 == 0)
            Console.println("Ping: pong")
          if (pingsLeft > 0) {
            pong ! Ping
            pingsLeft -= 1
          } else {
            Console.println("Ping: stop")
            pong ! Stop
            exit()
          }
      }
    }
  }
}
```

```
class Pong extends Actor {
  def act() {
    var pongCount = 0
    loop {
      react {
        case Ping =>
          if (pongCount % 1000 == 0)
            Console.println("Pong: ping "+pongCount)
          sender ! Pong
          pongCount = pongCount + 1
        case Stop =>
          Console.println("Pong: stop")
          exit()
      }
    }
  }
}
```

```
object pingpong extends Application {
  val pong = new Pong
  val ping = new Ping(100000, pong)
  ping.start
  pong.start
}
```

```
$ scalac pingpong.scala
$ scala -cp . examples.actors.pingpong
Pong: ping 0
Ping: pong
Pong: ping 1000
Ping: pong
Pong: ping 2000
...
Ping: stop
Pong: stop
```

Advantages of Erlang/Scala Model

- Weak coupling of processes via asynchronous and dynamically typed messages:
  - Expressive and flexible
  - Robust framework for error handling
  - Relative ease of upgrading running systems

- Potential disadvantage: checking happens at run time, so guarantees of robustness are probabilistic
  - Statically typed message passing systems like Singularity provide for compile-time checking that a process can respond to messages
  - Rendezvous-based synchronous systems provide better tests for liveness

Robust Message Passing Systems

- 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

http://akka.io/ for an alternative Scala actors library, implementing these fault tolerance concepts

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

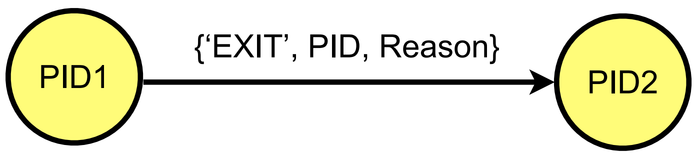
Let it Crash

- 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



Remote Error Handling: Advantages

- Remote error handling has several advantages:
  - “The error-handling code and the code which has the error execute within different threads of control
  - The code which solves the problem is not cluttered up with the code which handles the exception
  - The method works in a distributed system and so porting code from a single-node system to a distributed system needs little change to the error-handling code
  - Systems can be built and tested on a single node system, but deployed on a multi-node distributed system without massive changes to the code”

From: J. Armstrong, “Making reliable distributed systems in the presence of software errors”, PhD thesis, KTH, Stockholm, December 2003.

Erlang 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

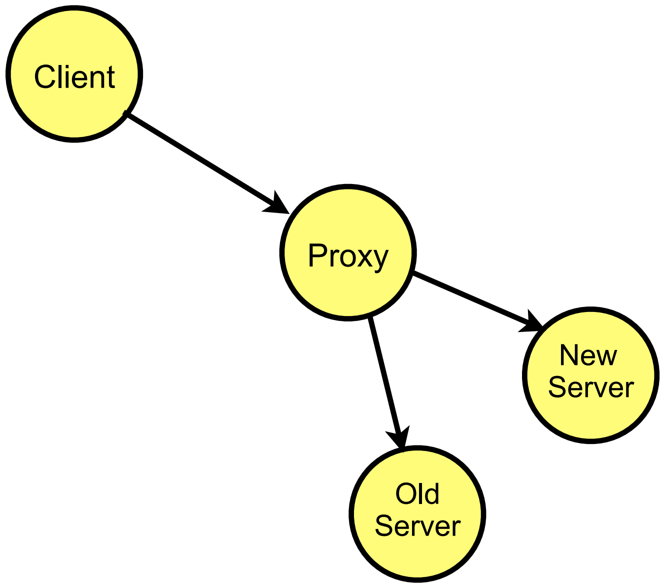
OTP: Open Telecom Platform – a library of useful behaviours for writing telecoms software

Erlang: Case Study

- Ericsson AXD301 160Gbps ATM switch
  - 1.1 million lines of Erlang
  - 2248 Erlang modules (equivalent to classes in an object-oriented system)
  - Dimensioned to handle ~50,000 simultaneous flows, with ~120 in setup or teardown phase at any one time
  - 99.9999999% reliable in real-world deployment on 11 routers at a major Ericsson customer (~0.5 seconds downtime per year)
  - Yet, process failures do occur, and are handled by the supervision hierarchy and distributed error recovery

Systems Upgrade and Evolution

- Message passing allows for easy system upgrade
  - Rather than passing messages directly to a server, pass them via a 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



Discussion and Further Reading

- J. Armstrong, “Erlang”, CACM, 53(9), September 2010, DOI:10.1145/1810891.1810910

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.” 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

- Discussion:
  - Is the Erlang approach to error handling appropriate, or is a statically typed system desirable?