Actor Based Programming

Paulo Sérgio Almeida

Grupo de Sistemas Distribuídos
Departamento de Informática
Universidade do Minho

Mainstream concurrency paradigms

- The concurrency paradigms used in the more popular languages (C, C++, Java, C#, . . .) are based on shared memory;
- Shared-memory concurrency:
    - is difficult and error prone (e.g., data races)
    - does not scale to many threads
- Event-based systems:
    - scale to many independent sources of events, but
    - also seen as difficult to program (inversion of control);
- Message based paradigms (CSP, CCS, Actors) used to be seen as “academic toys”.
    - Erlang and more recently Go changed this;

The problems with shared memory + monitors (I)

- Concurrency control difficult to program;
- broadcast / signalAll naive; optimizations may bring subtle bugs;
- Modern monitors feature an explosive mix:
  - absence of memory in signal;
  - state can change arbitrarily between signal and wait returning;
  - spurious wakeups;
- Shared memory does not scale to distributed systems;

The problems with shared memory + monitors (II)

- Systems with thousands of shared entities are difficult to program;
- Example: multi-player games;
- According to Tim Sweeney (Epic Games, Unreal Engine, POPL'06 talk) it is typical:
  - 30–60 updates per second;
  - 10000 objects active;
  - each object when updated affects 5 to 10 others;

  “Manual synchronization (shared state concurrency) is hopelessly intractable here”

The problems with shared memory + monitors (III)

- If it is difficult for the normal case. . .
- . . . it gets worse when things go wrong;
- An error in a thread may affect the whole system:
    - e.g., a thread raises an exception without releasing lock;
    - difficult to take care when many locks are involved;
- If it is unpleasant with only one lock . . .

```
public void m() {
  lock.lock();
  try {
    // ... method body
  } finally {
    lock.unlock()
  }
}
```

The problems with shared memory + imperative programming

- It is difficult to “undo” when things go wrong:
  - a thread wants to execute A and B;
  - B cannot be finished – e.g., due to unexpected exception;
  - effect of A in shared memory, will affect other threads;
- Problem with imperative paradigm (updating objects in place);
- Concepts like STM (software transactional memory) may help, if they can be efficiently implemented and become mainstream.

Actor model

- Introduced by Carl Hewitt in 1973;
- An actor is an entity that, in response to a message received, can concurrently:
  - send messages to other actors;
  - create a finite number of new actors;
  - choose behavior for processing next message;
- General and vague;
- Variants in use depart in different ways;

Actors – fundamental characteristics in practical variants

- Light-weight entity;
- Local encapsulated state;
- Asynchronous message passing;
- Sequential specification of local behaviour;

Light-weight entity

- An actor is an abstract light-weight entity;
- Unlike OS processes or threads;
- Can be created in large numbers, like objects;
- Very little creation and context-switch cost;
- Has a unique identity, like objects;

Local encapsulated state

- Each actor has local state + local behaviour;
- No other actor can access its state, unlike objects:
  - e.g., encapsulation breaches,
  - e.g., aliasing of reachable mutable objects;
  - made worse with threads+objects;
- All inter-actor communication is by messages;
- Can be seen as pure objects in the original sense (Alan Kay):
  - interacts by message passing;
  - encapsulates local state;

“The big lie of object-oriented programming is that objects provide encapsulation” *John Hogg*

Asynchronous message passing

- An actor can send messages to other actors;
- A message is an abstract immutable value;
- Actor identity used as destination;
  - no other concept, as channels, e.g., in CSP;
- Messaging by send and forget: no reply;
  - natural match to distributed systems;
  - if a ‘reply’ is needed, another message must be sent back;
  - contrary to objects: a method is invoked, runs and can give a result;
- All inter-actor communication is by messages;

Sequential specification of actor behaviour

- Original definition (everything concurrent) uneeded in practice:
  - actor creation fast (no need to wait for some initialization);
  - messages are just sent; no wait for reply;
- Logic of processing message received is purely sequential;
- No concurrent access to actor state;
- Responding to different kinds of message analogous to `case`;
- Concurrency arises in the system, due to many actors;

Message ordering

- Original definition: no assumption;
- Can complicate logic;
- In practice (most implementations) FIFO:
  - If A sends message m_1 and then m_2 to B, m_1 will be delivered first;
  - If actor A sends m_1 and actor B sends m_2 to same actor C, they can be delivered in any order;
  - FIFO is useful to programmers and cheap to implement;
- Stronger guarantees (e.g., causal order) usually not made:
  - If actor A sends m_1 to C, then m_2 to B and
  - after receiving m_2, B sends m_3 to C,
  - there is no guarantee about m_1 and m_3 delivery order;
- The Pony language guarantees causal delivery of actor messages.

Selective message processing

- Messages are usually stored in a mailbox in delivery order;
- When *A* is processing some request m_1 it may be useful to contact some other actor *B*:
  - sending some query to *B* in m_2;
  - *b* will ‘reply’ with some m_3;
  - it may be essential to process m_3, to be ably to ‘reply’ to original request, ignoring other messages in the mailbox;
- Being able to specify what messages to process is usually called *selective receive*;
- Allows ignoring other messages, as if they have not yet arrived;
- Allows encapsulating services (in libraries) with blocking APIs;
- Allows simplifying state-machines;

Actor as unit of isolation, ownership and fault-tolerance

- If a thread aborts and corrupts a shared data-structure:
  - it will become unusable by other threads;
  - no one owns the data, no other thread knows about the problem;
- An actor owns its state:
  - either a ‘client’ gets a ‘reply’ or none;
  - if there is a problem, it can be known whom to blame;
  - many times it can be solved by restarting the actor;

Let it crash philosophy and actor supervision

- Large systems contain bugs;
- System should work, even having some incorrect part;
- Erlang introduced supervision hierarchies:
  - an actor can be supervised by another actor;
  - if it crashes, its supervisor will receive a message;
  - the supervisor will act accordingly, possibly restarting the actor;
- Erlang popularized ‘let it crash’, advocating less defensive programming;

Case studies: actor model in practice

- Popularized by Erlang;
- Brought to the JVM by, e.g.:
  - Akka
  - Quasar
- Erlang has a VM suitable for it;
  - Other languages on that VM: e.g., Elixir, LFE;
- In the JVM more work needed to support it;

Case studies: Erlang

- Special purpose Erlang VM;
- Strict dynamically typed functional core;
- Light-weight processes;
- Preemptive: can have infinite loops in actors;
- Per-process stack, heap and garbage collection;
- Messages physically copied, even being functional language;
- Allows selective receive; easy to write with pattern matching;

Case studies: Akka

- Java library;
- No support for light-weight processes;
- Actor cannot block without blocking a thread;
- Leads to event-driven programming: a message is typically processed by a handler that runs to completion (until returning);
- Does not allow selective receive;
- Global heap, subject to long garbage collection pauses;
- Messages need to be made immutable by convention;
- Actor encapsulation needs some care;
- Waiting for reply uses other means such as futures;
- Need care in mixing constructs: e.g., a future callback runs in other thread and can result in races;

Case studies: Quasar

- Java library with bytecode instrumentation;
- Support for light-weight processes (fibers);
- Actors can block without blocking a thread;
- Allows selective receive;
- Global heap, subject to long garbage collection pauses;
- Messages need to be made immutable by convention;
- Actor encapsulation needs some care;
- Allows staying in pure actor model: no need to use other constructs, as in Akka;
- Supports other concepts: CSP channels and data-flow programming;