Assessing Design Patterns for Concurrency

Fikre Leguesse Adrian Francalanza
Department of Computer Science, ICT
University of Malta
{fleg0001,afra1}@um.edu.mt

Abstract

Design patterns are language-independent software-engineering techniques for solving recurring problems within a particular problem-context. Despite their generality, they have been primarily adopted by, and for the most part developed within, the object-oriented community. As a result, some pattern definitions are sometimes expressed with objects-oriented machinery in mind such as classes and inheritance.

We test the paradigm independence of these design patterns and investigate the viability of adopting existing patterns from the object-oriented paradigm to the message-passing concurrency setting. By porting these techniques to the new programming paradigm, we expect to inherit the benefits associated with these design patterns. We achieve these goals by implementing a suite of design patterns as reusable modules in Erlang, an industry-strength message-passing language for programming concurrent distributed systems.

Index Terms

Design Patterns, Concurrency, Message Passing, Shared Memory, Erlang

1. Introduction

Design patterns are solutions to recurring problems within a given context [1]. They provide documented techniques for solving re-occurring problems allowing for the reuse of quality design and time-tested standard approaches. Over the years the object-oriented (OO) community has successfully applied design patterns in the implementation of large scale software applications. These patterns have provided the community with a common problem solving mind set and a common vocabulary for the articulation and expression of architectural and design concepts for software solutions [1].

Despite this success within the OO community, they have not been exploited as extensively in other programming paradigm. It is often understood that design patterns are only applicable to OO design [2] and part of this misconception has been attributed to the extensive amount of material found in the OO community and the lack of related work in other fields. Furthermore, prominent material in the field [1], [3] tend to model patterns in terms of collaborating objects, structured using OO machinery such as classes and inheritance.

We concur with the commonly held view [2] that this should not be the case. Patterns capture expertise in a generic context. Since developers often encounter design problems that span across paradigms, it is paramount that these patterns are paradigm-independent, so as to allow problem solving expertise to transcend these paradigms.

The aims of this report are two-pronged:

1) We test the paradigm independence of a number of concurrency design patterns, by porting them from the shared-state OO concurrent model to the message-passing concurrent model, as this constitutes a substantial shift in programming paradigms. We choose an actor-based language [4] called Erlang[5] as our message-passing language, because of its maturity in terms of existing realistic Erlang code.
2) We explore the potential reuse of these patterns by expressing them as Erlang *behaviours i.e.*, higher-order code which teases apart the generic component of the patterns from the specifics of the pattern instantiation. This method of structuring code follows standard software design principles and is prevalent to OO pattern implementations through mechanisms such as inheritance and interfaces.

The task is non-trivial mainly because it involves a substantial switch in terms of the *units of decomposi-*

tion and *methods for entity interactions* of these design patterns. More specifically, in the OO case, these building blocks and glue tend to be objects, classes, packages, interfaces and abstract classes, method calls, field updates, monitors, inheritance and delegation, to name a few. In the message-passing paradigm, particularly in the Erlang actor model, we will instead need to use processes, modules and callbacks, higher-order functions, asynchronous messages and mailboxes.

Despite these immediate obstacles, there are a number of benefits we expect to gain from our endeavour:

Broadening Expertise Capture: The expertise we adopt, though already tried and tested [3], is at present mainly targeted, thus only accessible, to the OO community. Our work contributes towards *extending this expertise* to the message passing concurrency setting.

Standardizing Common Vocabulary: Across different paradigms, programmers often use different terms to refer to the same concepts and solutions. Our work helps towards standardizing these cosmetic differences by promoting a *common vocabulary*. Moreover, [6] argues that to provide a common vocabulary, *"there should be some restriction on the amount of design patterns in circulation, otherwise the social and cultural benefits tied to patterns become faint."* By reusing existing patterns rather than creating new ones, our work also satisfies this criteria.

Inducing Universal Descriptions: Through applying these patterns to a substantially different programming paradigm, we indirectly provide a higher-level description of these patterns, where explicit references to OO mechanisms are replaced with paradigm-independent ones. This disciplines pattern-designers to provide universal pattern definitions that are easier to comprehend.

Encourage Pattern Adoption: Design patterns promote the use of standard approaches and time-tested solutions to re-occurring problems. By structring them as behaviours, thus abstracting unnecessary details, we simplify their adoption and standardise code structure.

Design Reusability: Through behaviours, we also facilitate their reuse, because the generic part is clearly delineated and programmers need only recode the specific parts to reuse a pattern.

Expedite Development and Maintenance: Design patterns also provide maintainable solutions through decomposition into reusable components with high cohesion and low coupling. Clearly delineated components can be plugged into place and replaced with less effort and can be unit-tested in advance, which simplifies the discovery of errors during code development.

The rest of the paper is structured as follows. Sec. 2 outlines the design patterns we chose to port to Erlang. To illustrate our approach, in Sec. 3 we focus on one for these patterns, and show how this can indeed be expresses in a message-passing paradigm. In Sec. 4 we go one step further and restructure this pattern as a behaviour, also showing how this restructuring facilitates pattern reuse. Sec. 5 comments on these results and Sec. 6 concludes.

2. The Patterns

We have adapted five design patterns from [3], [1] to Erlang, all of which are considered to be concurrency design patterns. The decision to focus on concurrent patterns was motivated by the fact that, since Erlang targets concurrency, the eventual adaptations would turn out to me more natural, thereby giving us more ground for meaningful analysis.[1] The patterns are:

Active Object: In OO programming, this pattern is used to decouple the execution of a method from its invocation and advocates that each entity executes in its own thread of control, communicating using asynchronous message passing. It is based on the actor model for concurrency, which Erlang itself is based on, which made the adaptation straightforward.

Acceptor/Connector: This pattern targets client-server architectures where a server application must handle requests from a number of clients communicating over a network. It aims to decouple the connection and initialization stages from the application specific processing performed once a connection has been established which provides the flexibility for services to be added and removed transparently, without the need to reimplement connection-establishment code.

Observer: This pattern allows multiple clients to observe the state changes on a server without the need for the client to constantly poll the server. It provides an efficient and scalable design for decoupling clients from servers by having multiple observers registering themselves with a subject which automatically forwards any state changes to registered entities. The increase in efficiency stems from the inversion of control, placing the responsibility on the server rather than having multiple interested observers constantly checking with the server for any updates.

Proactor: It provides a strategy for concurrently handling asynchronous I/O events from multiple sources, by decoupling concurrent I/O events from concurrent processes. The pattern simplifies the dispatching of

1. We did in fact also implement more "traditional" patterns such as the *factory* and *singleton* patterns, but they gave us more insight into the functional aspect of the Erlang language rather than its message-passing aspect.

completion events from asynchronous operations by integrating the demultiplexing of completion events and the dispatching of their corresponding event handlers. A single proactor process is responsible for receiving completion events from some asynchronous event source, and spawning completion event handlers.
Leader/Followers: This architectural pattern provides an efficient concurrency strategy for threads or processes to coordinate themselves taking turns detecting, demultiplexing and dispatching events from a shared set of handles to the appropriate service request handlers. It aims to minimize overhead when creating and managing multiple processes sharing some resources. A process pool is created once and processes are recycled in order to minimize the overhead associated with creating/destroying processes. The processes coordinate themselves to ensure that only one process (the leader) uses the resources at any given time.

A full explanation of our work relating to all of these patterns can be found in [7]. Given the space limitations, in what follows we only focus on the Leader/Followers pattern.

3. Adapting Patterns to Message Passing

We outline how the Leader/Followers [8] pattern can be smoothly implemented in terms of Erlang processes. The pattern deals with three concurrency issues:

1) It provides a way of demultiplexing I/O events on a source (such as a TCP socket handle) to its appropriate service handlers.
2) it minimizes concurrency-related overhead by making use of a process pool
3) it prevents race conditions from occurring by coordinating the processes in the pool.

The pattern has three key participants: handles, service handlers, and a process pool. The process pool consists of a number of service processes that can be in one of three states: leading, processing, or following. Fig. 1 shows the possible process transitions. A leader process waits for an event to occur on some shared handle. Once an event is detected, the current leader promotes a new leader from the process pool, and becomes itself a processing process. Processing the event involves dispatching the appropriate service handler. Once processing is complete, the process becomes a follower once again, waiting to be promoted. In the special case where there is no current leader, the processing process skips directly to the leading state.

Fig. 2 shows the module implementing the server restricting concurrent active connections throught the Leader/Followers pattern. If connection requests are

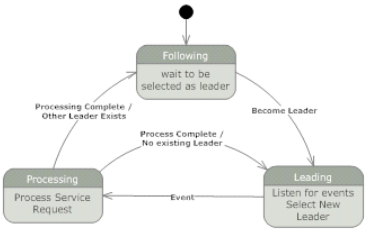

Figure 1. Process Transition for Leader/Followers

```
1  start(Port, PoolSize) ->
2      {ok, Handle} = gen_tcp:listen(Port, ?LISTEN_OPTS),
3      ProcessPool = create_processes(PoolSize, Handle, null),
4      [Leader | Followers] = ProcessPool,
5      Leader ! start_listening,
6      pool_manager(Followers).
7
8  pool_manager(ProcessPool) ->
9      receive
10         {select_next_process} ->
11             {ok, NewPool} = select_next_process(ProcessPool),
12             pool_manager(NewPool);
13         {add_process, Process} ->
14             {ok, NewPool} = add_process(ProcessPool, Process),
15             pool_manager(NewPool)
16     end.
17
18 process({PoolMgr, Handle, State}) ->
19     receive
20         start_listening ->
21             {ok, Socket} = gen_tcp:accept(Handle),
22             PoolMgr ! {select_next_process},
23             receive
24                 {tcp, Socket, Bin} ->
25                     log(Bin),
26                     timer:sleep(?DELAY),
27                     gen_tcp:send(Socket, Bin),
28                     gen_tcp:close(Socket);
29                 {tcp_closed, Socket} ->
30                     ok
31             end,
32             PoolMgr ! {add_process, self()},
33             process({PoolMgr, Handle, State})
34         end.
```

Figure 2. Echo Server (Leader/Followers).

received faster than the pool processes can handle them, then the requests are queued up on the TCP socket until an idle process is placed in the pool. The application is a simple echo server, accepting connections from multiple clients, echoing any data received over the socket back to the client.

This module consists of three main functions (some local functions omitted). The first two, `start/2` (line 1) and `pool_manager/1` (line 8), form the pool manager process. The `start/2` function provides the initialization code while `pool_manager/1` provides its recursive loop. The third function `process/1` (line 18) describes a single process within the pool.

The `start/2` function initializes the handle shared by the processes, *i.e.*, a TCP socket listener (line 2), and spawns the processes forming the pool (line 3). The `create_processes/3` function is omitted and takes the size of the pool (N) and the shared handle as input parameters, spawning N processes which start as followers. The `start/2` function then selects a process from the pool (line 4) and promotes it as leader by

sending it a `start_listening` message (line 5). This function then calls the `pool_manager/1` function (line 8) which provides the pool manager's recursive loop.

The role of the pool manger is to coordinate the processes within the pool. When it receives a `select_next_process` message (line 10), it calls function `select_next_process/1` (line 11) which selects a pool process, notifying it that it has been promoted as leader. In the case where the pool is empty, the request is ignored.

When the `pool_manager` process receives an `add_process` message (line 13), it calls the `add_process/2` function (line 14). This adds the process to the pool. In the case where there is no current leader, the process is promoted to leader rather than being added to the pool.

The function `process/1` (line 18) implements a single process within the process pool. This process goes through a number of cyclic steps:

1) Once spawned, the process blocks on the receive statement (line 19) waiting to be promoted as the new leader by receiving a `start_listening message` (line 20).
2) Once the message is received, the process begins to listen for events on the shared handle, *i.e.,* accept events on the TCP socket (line 21).
3) Once an event occurs, the pool manager is notified that it should promote a new leader (line 22) to listen for events on the handle.
4) The process then starts processing the event by accepting some data over the created socket (line 24), logging the data and sending it back to the client (lines 25-27).
5) Once processing is complete, the process notifies the pool manger that it is ready to listen for other events (line 32) and is placed in the process pool waiting to be promoted once again.

The immediate benefits gained from adopting the Leader/Followers pattern for dealing with multiple clients are that, through standardized common vocabulary and familiar higher-level code structures, the overall architecture of the code is easer to comprehend. Moreover, with the exception of the socket handle initialization, the functionality provided by the pool manager is generic and is common to any application implementing this pattern. This functionality can therefore be abstracted away into a reusable module as will be discussed in Sec. 4.

4. Patterns as Behaviours

We split the implementation of the Leader/Followers server into two modules, teasing apart the generic aspects of the pattern from the specific aspects of the pattern instantiation. Abstracting the generic part of the Leader/Follower into a single reusable module then permits any application following the pattern to simply extend this module.

The module encapsulating the generic operations of Leader/Followers is called the *behaviour module*. The client module using the behaviour is called the *callback module*, and provides the application specific aspects of the pattern which the behaviour delegates back to when required, in the form of requests to the callback functions. The behaviour is thus implemented using higher-order functions which take the callback module's name as an input parameter as shown in line 1 of Fig. 3.

As discussed in Sec. 3, the functionality provided by the function `pool_manager/1` is common to all servers implementing the Leader/Followers pattern. Only the initialization of the handle is specific to the application implementing the pattern and is therefore the responsibility of the callback module. As for the processes within the pool *i.e.,* `process/3`, steps 2 and 4 (listening for events and processing the event) are application specific whereas all other steps can be captured by the behaviour module.

Fig. 3 shows the `gen_leader_follower` behaviour (with some local functions omitted). This behaviour defines the pool manager while also providing the skeleton for a pool process. The `start/4` function and the `pool_manager/1` function (lines 1 and 7), which make up the pool manager, are almost identical to Fig. 2, the only difference being that the shared handle is initialized by the callback module. This is passed to the function as a parameter (line 1) as it is application specific.

Callback modules extending `gen_leader_follower` behaviour with specific behaviour need to implement the following callback functions used by the pool process:

- `listen_for_events/2`: returns `{ok, Event}`. This is a blocking function that is called by the leader process. Its role is to listen for an event on a given handle. The function returns with the received Event.
- `handle_events/3`: returns `any()`. This function is called on to process the events received by the process. It provides application specific services.

Pool processes are spawned by the `pool_manager` process on initialization with the callback module passed as a parameter. The pool process called by the spawning process `process/1`, Fig. 3 (line 17), in the `gen_leader_follower` behaviour, functions as follows:

```
1  start(Mod, Handle, PoolSize, State) ->
2      Pool = create_processes(PoolSize, Mod, Handle, State),
3      [Leader | Followers] = Pool,
4      Leader ! start_listening,
5      pool_manager(Followers).
6
7  pool_manager(ProcessPool) ->
8      receive
9          {select_next_process} ->
10             {ok, NewPool} = select_next_process(ProcessPool),
11             pool_manager(NewPool);
12         {add_process, Process} ->
13             {ok, NewPool} = add_process(ProcessPool, Process),
14             pool_manager(NewPool)
15     end.
16
17 process({Mod, PoolMgr, Handle, State}) ->
18     receive
19         start_listening ->
20             {ok, Event} = Mod:listen_for_events(Handle, State),
21             PoolMgr ! {select_next_process},
22             Mod:handle_events(Event, Handle, State),
23             PoolMgr ! {add_process, self()},
24             process({Mod, PoolMgr, Handle, State})
25         end.
```

Figure 3. gen_leader_follower behaviour.

1) Initially, it blocks on the receive statement (line 18) waiting to be promoted.
2) Once promoted, the process calls its callback modules function `listen_for_events/2` (line 20) which returns with the event fired on the handle.
3) The pool manager is then asked to promote a new leader (line 21) while it handles the event by dispatching to the service handler, `handle_events/3`, also defined in the callback module (line 22).
4) Once processing is complete, the process notifies the pool manger that it is ready to listen for other events (line 23), and is placed with the followers in the process pool.

This behaviour captures the essence of the Leader/Follower pattern. It can be analyses and tested as a unit, independent of the application specific details to be defined by the callback module. This organisation facilitates design and code reusability. To illustrate this point, we provide two examples of callback modules with differing application specific functionality.

The first example callback module is shown in Fig. 4, and uses the `gen_leader_follower` behaviour to provide an implementation of the echo server we discussed earlier in Fig. 2. Fig. 4 shows the two required callback functions, as well as a `start/1` function. The `start/1` function (line 1) initializes the TCP listener handle (line 2), and calls the `start/1` function on the behaviour module (line 3) passing the callback module's name (represented by the macro MODULE) as the callback module. The function `listen_for_events/2` (line 5) is called on by the leader process in order to detect a fired event on the given handle. The function `handle_events/3` (line 9)

```
1  start(Port) ->
2      {ok, Listen} = gen_tcp:listen(Port, ?LISTEN_OPTS),
3      gen_leader_follower:start(?MODULE, Listen, 5, null).
4
5  listen_for_events(Handle, _Counter) ->
6      {ok, Socket} = gen_tcp:accept(Handle),
7      {ok, {ok, Socket}}.
8
9  handle_events({ok, Socket}, _Handle, _Counter) ->
10     receive
11         {tcp, Socket, Bin} ->
12             log(Bin),
13             timer:sleep(?DELAY),
14             gen_tcp:send(Socket, Bin),
15             gen_tcp:close(Socket);
16         {tcp_closed, Socket} ->
17             ok
18     end.
```

Figure 4. Echo server - gen_leader_follower callback module.

```
1  start(Port) ->
2      {ok, Listen} = gen_tcp:listen(Port, ?LISTEN_OPTS),
3      gen_leader_follower:start(?MODULE, Listen, 5, null).
4
5  listen_for_events(Handle, _Counter) ->
6      {ok, Socket} = gen_tcp:accept(Handle),
7      {ok, {ok, Socket}}.
8
9  handle_events({ok, Socket}, _Handle, _Counter) ->
10     receive
11         {tcp, Socket, _Handshake} ->
12             Time = erlang:localtime(),
13             FormatedTime = format(Time),
14             log(FormatedTime),
15             gen_tcp:send(Socket, FormatedTime),
16             gen_tcp:close(Socket);
17         {tcp_closed, Socket} ->
18             ok
19     end.
```

Figure 5. Time server - gen_leader_follower callback module.

is called by the same process once a new leader has been promoted. This function receives data over the socket, sending it back to the client.

The code for this module is significantly shorter and easier to comprehend than the initial implementation of Sec. 3. In fact, the original implementation (including the omitted functions) spanned over 52 lines of code, whereas the implementation using the behaviour is simply 25 lines of code (including the missing directives).

Apart from shortening development times through shorter code, the `gen_leader_follower` behaviour provides more scope for reusability of pre-tested code. In fact, implementing a new server process following the Leader/Followers pattern simply involves creating a new callback module. Figure 5 shows another server application that implements the pattern.

This particular server allows clients connected over a TCP network to request the current time. Once a TCP connection is established (line 6), the client sends a handshake (line 11). The server then sends the current date and time over the TCP network to the client. The use of the process pool ensures that the number of

concurrent clients is limited to a predefined number.

Comparing this code with the echo server module, changes are made in the function `handle_events/3` (line 9) in which the current time is determined, formatted, and sent to the client (lines 12-15). The use of the behaviour considerably reduces the amount of code that needs to be developed. Had we not used the `gen_leader_follower` behaviour, we would have been required to implement the pool manager once again, as well as the processes' protocol. Moreover, with this new code organisation, the `gen_leader_follower` behaviour abstracts the pool and process management code away from the programmer.

5. Results

We have outlined through the Leader/Followers pattern how we adapted concurrency patterns that were originally designed in terms of passive objects communicating through method calls which are synchronised using object monitors. Instead we implemented these patterns using *processes* as our main unit of decomposition and entities communicate asyncronously using message passing and mailboxes. We substituted OO mechanisms such as inheritance with delegation, higher order functions, and polymorphism through the use of first class functions, which are applicable to Erlang and other message-passing concurrent languages. In addition, we use the module callback mechanisms in Erlang to structure patterns as behaviours, thereby promoting modular design, software abstraction and code reuse.

In [7], we also tested the integration and usability of the patterns through the implementation of a peer-to-peer file sharing application, which requires a high degree of concurrency to cater for multiple concurrent clients while sharing files broken down into numerous pieces. Figure 6 shows the patterns used by the peer-to-peer application. The columns show the five design patterns implemented as behaviours, while the rows show the processes involved in the application. The entries in the table show the role of each participant in the design pattern it implements. These either extend a behaviour or contribute in some way to the patterns' protocol.

Process	Active Object	Acceptor - Connector	Observer	Proactor	Leader-Followers
Manager	actor				
Acceptor				initiator	Process
Peer group mgr			observer	proactor	
Peer recv		connector			
Peer Send	actor	srvc hndlr			
tracker comms		connector	subject		
file system	actor				

Figure 6. behaviours/patterns in the peer app.

6. Conclusion

We have shown that concurrency design patterns targetted to an OO paradigm can be feasibly adapted to a message passing concurrency setting. Within this setting, we found natural units of decomposition for these patterns as well as mechanisms for facilitating modular design, encapsulation and code reuse. In fact, in some cases, the adaptation of these patterns proved to be more of a natural fit to a message-passing process model as opposed to a shared-state model with passive objects. As future work we plan to exploit the modular design of these behaviours to devise incremental testing procedures using tools such as QuickCheck[9].

References

[1] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] J. Vlissides, *Pattern Hatching: Design Patterns Applied*, ser. Software Patterns Series. New York, NY: Addison-Wesley, 1998.

[3] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects*. New York: John Wiley and Sons, 2000.

[4] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: The MIT Press, 1986.

[5] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[6] E. Agerbo and A. Cornils, "How to preserve the benefits of design patterns," in *OOPSLA 1998*, Anaheim, California, 1998.

[7] F. Leguesse, "Assessing design patterns for concurrency," CS Dept., ICT, University of Malta, Tech. Rep., 2009.

[8] D. Schmidt and C. O'Ryan, "Leader/Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching," in *Proceedings of the 6th PLoP*, Monticello, IL, 2000.

[9] K. Claessen, M. Pałka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger, "Finding race conditions in erlang with quickcheck and pulse," in *Proceedings of the 14th ACM SIGPLAN ICFP '09*. New York, NY, USA: ACM, 2009.