What is Akka?

Large-scale applications that depend on a complex programming model are difficult to maintain and reason about; simplicity is key to success. Applications are deployed more often across many servers, and there is a need to get these applications done simpler and quicker, using one programming model.

More and more applications need to integrate with real-time or near real-time information. The services that provide this information are somewhere on the Internet, and can be unavailable, unresponsive or misbehaving for many reasons. The end-user expects a responsive experience, even if some services are not available, across a range of devices. Calling out to every service synchronously and blocking for every response is not an option in this case.

The rise of cloud service providers in the past years shows that there is a real need to deploy distributed applications on scalable infrastructure. To make proper use of these cloud services, applications obviously need to be more scalable and distributed.

To briefly summarize, there are a couple of items on our wish list:

1. Handling many requests in parallel
2. Concurrent interaction with services and clients
3. Responsive asynchronous interaction
4. An event-driven programming model
5. Scaling out when needed
6. Increased end-user expectations of performance and availability
7. Fault tolerance across several nodes in a distributed system
8. Running on elastic cloud platforms, instead of statically clustered environments

As an example, at CSC we developed a sensor grid system to assist the border police. It automatically recognizes vehicles crossing the Dutch borders at fixed highway locations and from within patrol cars geared with sensors.

In this system, we had to combine real-time events from cameras and radars to one physical vehicle event. The sensor data is merged and analyzed to provide information about the vehicle like the license plate, color, speed, brand, and vehicle type. This information is sent to a central system and used to arrest suspects at the border in real time, so it had to be fast.

For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/roestenburg/
Akka simplified the task of combining the signals from sensors. The application does not block and wait for signals; it combines them as they arrive. We started out with one chained list of actors as an image-processing pipeline to do the image processing. When we found out that we could handle more images per second, it was very easy to scale up the amount of actors by using more processing pipelines in parallel.

Remote actors were used to build a fault tolerant messaging mechanism so that the system could operate efficiently over wireless UMTS connections. Remote actors also simplified the registration of new sensor nodes when the next highway at the border was deployed or when a patrol car was added.

The Akka-Camel module that provides connectivity to many protocols out of the box simplified integration with both hardware sensors and backend systems we had to integrate on a range of protocols, like HTTP, ActiveMQ, FTP and custom TCP/IP protocols.

If you’re building a system that has a wish list that is comparable to the one shown above, Akka could be a good fit. In this green paper, we’re going to show you the main features that set Akka apart from the conventional frameworks and libraries that you are probably familiar with. First, we will look at what Akka consists of. After that, we will look at Akka’s main features from a conceptual level, starting off with how Akka provides a simpler model for concurrency. After that, we will find out that this simpler model has a couple of benefits that can be used to build more fault tolerant and more scalable systems.

**The Akka toolkit**

Akka is a toolkit and takes a different approach than conventional frameworks. Naming it a toolkit instead of a framework has been a very distinct choice of words by the Akka team. Think of it as a tool belt, where you carry a couple of tools with you, you don’t necessarily use all of the tools, but your most used tools are right there close at hand. You might keep other tools in your toolbox because they are somewhat heavy to wear in the tool belt or you might hardly use them. It really depends on the job at hand. You use the tools when you need them. Akka has been built from the ground up with testing in mind. Every aspect of it is easy to start up and test within your own code.

Akka is made up out of modules that are distributed as JAR files; you use them just like any other library. Great care has been taken to minimize the dependencies that are needed in every module. The Akka-actor module has no dependencies other than the standard Scala library and is in fact distributed with Scala as from version 2.10. There are core modules available for remoting, clustering, transactions, and dataflow concurrency, while other modules are focused on integration with other systems like the Camel and ZeroMQ module.

Akka also provides a runtime. The runtime provides shared services and configuration of all the modules that you would like to use. There is a microkernel that you can use to deploy your applications. So what are the main elements of a typical Akka application? Well, the answer is, actors. The central concept in Akka is the actor programming model. As a first definition, actors are the asynchronous building blocks that simplify the building of concurrent and distributed applications. We are now going to look at how the actor programming model is a simpler model for concurrency.

**Simpler concurrency**

The next example application needs to handle some requests concurrently. A concurrent application consists of several activities that need to execute simultaneously and at the same time interact with each other to complete the task at hand. We’re expecting that you have at least some experience with writing concurrent code using threads on the JVM and have experienced some of the hard problems that come with the territory. So here’s to hoping you find it as hard as we do and would love a simpler model than threads and locks for concurrency, and as an added bonus, a lot less code to get done. In this section, we are going to look at concurrency from a conceptual level and look at different ways to solve a problem—selling tickets.

**The example problem: selling tickets**

In this example, customers buy tickets to sports games from kiosks. The following figure shows a customer buying a ticket.

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/roestenburg](http://www.manning.com/roestenburg)
Many customers can buy tickets at many kiosks at the same time. The tickets are provided to the kiosks by a printing office. Tickets are printed in batches to minimize the amount of printed tickets that might never get sold. This is where the application becomes concurrent, since the kiosks interact with the printing office while they sell tickets to many customers simultaneously. The goal is to be able to sell as many tickets as possible at the same time. Figure 2 shows the relationship between the customers, kiosks, and the printing office.

There are a couple of rules that need to be followed:

- A game takes place only once at a particular sports stadium. The maximum amount of tickets that can be sold is limited to the amount of seats in the stadium. The printing office can only print tickets up to the same amount.
- A ticket is only valid for one particular game.
- The same ticket cannot be sold to more than one customer. Customers cannot share tickets.

A game can only be in one of the following states:

- The game has tickets for sale. The printing office has not printed the maximum amount of tickets yet. The game is not sold out yet.
- The game is out of print, meaning that the maximum amount of tickets has been printed by the printing office. There are still tickets available for sale. The game is not sold out yet.
- The game has no more tickets for sale and is out of print. The game is sold out.

In the next two sections, we will look at two different ways to solve this problem.

**Option 1: sharing state approach**

One way to communicate the state of a game and its tickets is to share some data structure. Both the kiosks and the printing office can reference a shared list of sports games. Every game has a list of tickets. The kiosk can

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/roestenburg/](http://www.manning.com/roestenburg/)
remove tickets from this list when it sells tickets; the printing office can add tickets to this list when it has printed some new ones.

This list can be changed by both the kiosks and the printing office at any time. We will call this the *shared mutable state* approach to concurrency. Figure 2 shows how the kiosk interacts with the shared list when a customer buys a ticket.

![Diagram](image)

The kiosk takes a ticket (3a) and requests more tickets to be printed when the tickets are running low (3b). If the game is out of print, and there are no more tickets, the kiosk sets the game to sold out (3c).

**Figure 3 Buying sequence**

Whenever a customer buys a ticket, the kiosk takes a ticket from the game. If the tickets for the game are running low and it is not out of print yet, the kiosk requests new tickets from the printing office. Figure 4 shows how the printing office interacts with the shared list of games.
The printing office initially adds a game (1) and adds the first tickets to that game (2).

Figure 4 Printing sequence

The printing office initially adds games to the sports games list. Whenever a kiosk asks for new tickets, the printing office adds new tickets to the game or marks the game as out of print. As figure 5 shows, the Game and Sports Games classes need to be able to handle a couple of requests at the same time.

Figure 5 Concurrent access
As you see in the figure, you need to add a lock to the shared list so that we don’t get unexpected problems with multiple threads. Figure 6 shows some of the consequences of using locks.

When a kiosk waits for a lock, the customers at that kiosk have to wait. Dashed arrows wait while solid arrows have the lock in below figure.

The kiosks have to wait for each other at times, which means that the customers have to wait longer for their tickets. The kiosks also have to wait when the printing office is busy adding tickets. Some of these wait times just feel unnecessary. Why would a customer at one kiosk have to wait for a customer at another kiosk?

There might be a way to optimize this. Locking is often done too much or too little; when too many locks are used, nothing really happens at the same time. Difficult bugs can occur when too little locks are used. Getting to the right balance is hard.

Most problems that can occur with a shared data structure have to do with the fact that there is no guaranteed order at which the kiosks and the printing office interact with the shared list. The order is not the same every time and is dependent on the thread scheduling algorithm of the JVM, how long the action takes to complete, and how fast the shared list is accessed repeatedly by the objects.

Without locks, both kiosks could potentially sell the same ticket or try to remove a ticket that is not there anymore. With locks, if kiosks take tickets extremely often, the printing office might never get a chance to get in between. If the kiosk and the printing office access the list in the wrong order, they might end up waiting for each other forever.

These are examples of problems that are known as a race condition, thread starvation, and deadlock. Figure 7 shows the printing office and the kiosk in a deadlock situation.
For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/roestenburg/
We will call this approach the message-passing approach, as opposed to the earlier example of a shared mutable state approach. Instead of sharing a data structure, every kiosk will keep its own batch of tickets per game.

The buying sequence at the kiosk is almost the same as before except for that it does not share the list and it sends a MoreTickets message to indicate to the printing office that it needs more tickets.

So how do the kiosks get tickets? Well, the printing office will send messages to the kiosks. Every game message will contain a batch of tickets and some stats for every kiosk. There are quite a few ways to send tickets to all the kiosks; in this case, we have chosen to let the kiosks relay the game message to each other and build up some information for all the kiosks while the message is relayed. The kiosks know each others addresses and send the game message along to each other, as shown in figure 9.
For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/roestenburg/

The AKKA approach

As you’ve probably guessed already, the message-passing approach is the approach that Akka takes. The kiosk and the printing office in the message-passing example can be implemented with Akka actors. Actors do not share state, can only communicate through immutable messages, and do not talk to each other directly but through actor references, similar to the addresses we talked about. This matches exactly with the three things we wanted to change. So why is this approach simpler compared to the shared mutable state approach?

- We don’t need to manage manual locks. We don’t have to think about how to protect the shared data. Inside an actor, we’re safe.
- We are more protected from deadlocks caused by out-of-order access by several threads, which cause the system to wait forever or other problems like race conditions and thread starvation. We depend on the Akka toolkit to prevent most of these problems once and for all, instead of having to deal with it ourselves every
Performance tuning a shared mutable state solution is hard work and error prone.

The actor model is not new

The message-passing approach that we have loosely defined so far is also known as the actor model. The actor model is not new at all and has actually been around for quite a while. The idea was introduced in 1973 by Carl Hewitt, Peter Bishop, and Richard Steiger. The Erlang language and its OTP middleware libraries, developed by Ericsson around 1986, support the actor model and have been used to build massively scalable systems with requirements on high availability. An example of the success of Erlang is the AXD301 switch product, which achieves a reliability 99.9999999%, also known as nine nines reliability.

The actor model implementation in Akka differs in a couple of details from the Erlang implementation but has definitely been heavily influenced by it and shares a lot of its concepts.

So are there no concurrency primitives like locks used at all in Akka? Well, of course there are; it’s just that you don’t have to deal with them directly. Everything still eventually runs on threads and low-level concurrency primitives. Akka uses concurrency techniques from the java.util.concurrent library to coordinate message processing and takes great care to minimize the amount of locks used to an absolute bare minimum and uses lock free and wait-free algorithms where possible instead of using compare-and-swap (CAS) techniques. And, because nothing can be shared between actors, the shared locks that you would normally have between objects are not present at all. But, we can still get into some trouble if we accidentally share state. We also saw that in the message-passing approach we needed to find a way to redistribute the tickets. We had to model the application differently, which is what you would probably expect—no such thing as a free lunch.

There are other benefits that stem from the message-passing approach that Akka uses to its advantage. We have touched on them briefly already:

- The system does not stop functioning completely when one part of it crashes. The kiosks could continue to sell tickets even if the printing office crashed, sharing tickets with each other until their collective amount of tickets run out. It seems that the message-passing approach is more fault tolerant and can keep running in the face of serious problems.
- The shared mutable state is always in one place in the example (in one JVM if it is kept entirely in memory). If you need to scale beyond this constraint, you will have to (re)distribute the data anyway. Since the message-passing style uses addresses, there is probably an easier way to scale out if these addresses can also be used for remote servers.

Fault tolerance

This section is briefly going to show how the flexibility of the message-passing style can be used to achieve a more fault tolerant application. The tickets selling example briefly mentioned that the message-passing approach allows the system to keep functioning when part of it is waiting forever and not functioning at all. One reason for this is isolation is that the actors do not talk to each other directly. An actor can never block or wait forever because it sent another actor a message. The message is delivered to a mailbox. Maybe the message will never reach the other actor or maybe it will never get processed, but the sending actor at least will never fail because it’s trying to send a message to an address. The same cannot be said for objects and calling methods. Once you call a method, you are all in. The following figure shows the difference in this regard between actors and objects.
For source code, sample chapters, the Online Author Forum, and other resources, go to http://www.manning.com/roestenburg/
fault tolerance strategy that Akka provides, which is called the restart strategy. Other strategies that can be used are resume, stop, and escalate. Akka provides a way to select strategies for specific exceptions that can occur in actors. Since Akka controls how all messages between actors are processed and knows all the addresses of the actors, it can stop processing messages for an actor that throws an exception, check which strategy should be used at the specific exception, and take the required action.

Fault tolerant does not mean that every possible fault is caught and recovered from completely. A fault tolerant system is a system that can at least contain and isolate faults in specific parts of the system, averting a full system crash. The goal is to keep the system running, as achieved by restarting the printing office.

Different faults will need different corrective strategies. Some faults can be solved by restarting a part of the system, other faults might not be solvable at the point of detection and need to be handled at a higher level, as part of a larger subsystem.

As you would probably expect, replacing malfunctioning objects in a shared mutable state approach is almost impossible to do, unless you are prepared to build a framework around your code to support it. And this is not limited to malfunctioning objects; what if you would just like to replace the behavior of a particular object? (Akka also provides a functionality to hot swap the behavior of an actor.) Since you do not have control over how methods are called, or the possibility to suspend a method and redirect it to another new object, the flexibility that is offered by the message-passing approach is unmatched. Without going into a lot of detail, let’s look briefly at exception handling. Exceptions in standard, non-concurrent code is thrown up the call hierarchy. An object needs to handle the exception or throw it up the call stack.

Whenever an exception occurs, you need to stop the regular business that you are doing and fallback to error handling, to immediately continue where you left off after the error has been dealt with. Since this is quite hard, most developer prefer to just throw an error all the way up the stack, leaving it up for some type of framework to handle, aborting the process at the first sign of an error.

And that’s just talking about non-concurrent exception handling. Exceptions are almost impossible to share between threads out of the box, unless you are prepared to build a lot of infrastructure code to handle this. Exceptions cannot automatically be passed outside of the thread group that the thread is part of, which means you would have to find some other way to communicate exceptions amongst threads in different thread groups. In most cases, if a thread encounters some kind of exception, the choice is made to ignore the error and continue or stop execution completely, which is the simplest solution. You might find some evidence in a log that a thread crashed or stopped, but communicating this back to other parts of the system is not that easy. Restarting the thread and providing the state it needs to function is very hard to do manually. Things become an order of magnitude harder when these threads are distributed across several machines. Akka provides one model for handling errors, for both actors on one machine, as scaled out across many.

**Scaling up and out**

In this section, we are going to briefly look at how a message-passing style provides some benefits for scaling up and out. To scale up means to improve the performance of one server by adding resources to it, for instance, adding CPU cores. To scale out means to add more servers to a network and distribute the load across the servers so that the performance of the total system is increased.

**Scale out**

Without explicitly stating it, so far we have looked at the kiosks and printing office example on one machine, and one JVM. So how do we scale out to many machines?

The message-passing style uses addresses to communicate. So, all we need to change is how the addresses are linked to actors. If the toolkit takes care of the fact that an address can be local or remote, we can scale the solution just by configuring how the addresses are resolved. The following figure shows how the toolkit could solve this for us.

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/roestenburg](http://www.manning.com/roestenburg)
Akka implements this in a feature called remote actors. Akka passes messages for a remote actor onto a remote machine where the actor resides and passes the result back across the network. The kiosk does not know that it just talked to a remote printing office; it just interacted with the address. The solution can stay almost the same as the in-memory example. The only thing that has to change is how the reference to remote actors is looked up, which can be configured. The code can stay exactly the same.

The flexibility of resolving an address is heavily used in Akka. Remote actors, clustering, and even the test toolkit use this flexibility.

The shared mutable state example uses locks that only work within one JVM. So how easy is it to scale out and make the shared mutable state example work across many machines? There are no features in the JVM that we can directly use to achieve this. The most common solution is to push all of the state out of the JVM and put it all in a database.

In this case, the code has to be changed completely. First of all, the kiosk can’t just call “Find game” on the sports games structure, for instance, because it now only exists in the database. The locks will now have to be implemented in the database to keep the shared list consistent and avoid problems like deadlocks, which can still occur. You would need to find a way to cluster the database since otherwise you would have a single point of failure. The problem is pushed down to the database to solve. And, a database cluster does a lot more than our simple application needs; we might be paying a big overhead price. The kiosk requested more tickets directly from the printing office in the in-memory example, which is not possible anymore. The kiosk and printing office would have to communicate through the database in some way or through some other mechanism like web services. It seems that this is getting complicated quite quickly for what seems to be a simple requirement.

**Scale up**

So what about if we want to just increase the performance on one machine and scale up? Imagine that we upgrade a machine’s amount of CPU cores. In the shared mutable state example, we could increase the amount of threads that the kiosks run on. But, as we have seen, locks can prevent how many threads can execute at the same time, some will have to wait on each other to finish, so it has to be seen how much of a difference the increase of the amount of threads makes. Sharing as little as possible means locking as little as possible, which is the goal of the
message-passing approach. The below figure shows the kiosks and printing office sharing information about the sports games, using the shared mutable state approach and the message-passing approach. To highlight the difference, the shared mutable state example is shown in a worst-case scenario where every kiosk and the printing office have to wait on each other.

![Shared mutable state approach](image)

**Shared mutable state approach**

Kiosk 1, 2 and the printing office access the shared sports games twice through a shared lock. The blocks show the time that the sports games is used, the shaded blocks indicate the time that a kiosk or the printing office needs to wait for a lock. The below is a worst case situation.

<table>
<thead>
<tr>
<th>Kiosk 1</th>
<th>Kiosk 2</th>
<th>Printing Office</th>
<th>Shared list</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>K1</td>
</tr>
</tbody>
</table>

![Message passing approach](image)

**Message passing approach**

Kiosk actors 1, 2 and the printing office actor relay two game messages (g1, g2), for which new immutable copies are created and passed through (indicated by ‘ ’ and ‘ ‘). Every actor can start processing the moment it receives a message. The actors never block or wait.

<table>
<thead>
<tr>
<th>Kiosk 1</th>
<th>Kiosk 2</th>
<th>Printing Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>g1</td>
<td>g1'</td>
<td>g1''</td>
</tr>
<tr>
<td>g2</td>
<td>g2'</td>
<td>g2''</td>
</tr>
</tbody>
</table>

Figure 13 More threads

The above kiosks and printing office in the message-passing approach can run on fewer threads and still outperform the multi-threaded locking example, as long as the toolkit optimizes the processing and dispatching of messages and we compare with the worst case. Every thread has a stack to store runtime data. The size of the stack differs per operating system; for instance, on the Linux x64 platform, it is normally 256 KB. The stack size is one of the factors that limit the amount of threads that run at the same time on a server. Around 4096 threads can fit in 1 GB of memory on the Linux x64 platform.

Akka is easy to configure when it comes to tuning performance. Actors run on an abstraction, which is called a dispatcher. The dispatcher takes care of what kind of threading model is used and processes the mailboxes.

Actors are lightweight because they run on top of dispatchers; the actors are not directly related to the number of threads. Akka actors take a lot less space than actors; around 2.7 million actors can fit in 1 GB of memory. A big difference compared to 4096 threads, which means that you can create different types of actors more freely than you would use threads directly. There are different types of dispatchers to choose from which can be tuned to specific needs. It’s possible to share a dispatcher between actors or use different ones for different actors. Being able to configure and tune the dispatchers and the mailboxes that are used throughout the application gives a lot of flexibility for performance tuning.

For source code, sample chapters, the Online Author Forum, and other resources, go to [http://www.manning.com/roestenburg/](http://www.manning.com/roestenburg/)
Summary
You have learned what Akka is and what kind of problems it is trying to solve. Akka has made the choice to choose a message-passing style approach, which simplifies some of the hard problems of shared mutable state concurrency. We looked at the message-passing style from a conceptual level and introduced the concept of actors. Actors communicate with each other through sending immutable and asynchronous messages, not directly but through actor references. This approach is simpler because we don’t have to deal with locks and other low-level concurrency primitives directly. These properties provide a lot of flexibility compared to normal objects. The isolation provided by actor references and mailboxes makes it possible to choose a different approach to error handling and lays the foundation for a fault tolerant architecture. Several fault recovery strategies can be applied when parts of an actor system fail, isolating faults while the rest of the system can continue to operate. It also makes it possible to transparently communicate with remote actors across servers in a network. This simplifies how solutions can be scaled out.
Here are some other Manning titles you might be interested in:

- **Scala in Depth**
  Joshua D. Suereth

- **Scala in Action**
  Nilanjan Raychaudhuri

- **Play for Scala**
  Peter Hilton, Erik Bakker, and Francisco Canedo

Last updated: September 25, 2012