Clojure core.async Channels

- **Speaker: Rich Hickey**
- **Conference: Strange Loop 2013 - Sept 2013**
- **Video: http://www.infoq.com/presentations/clojure-core-async**

[Time 0:00:00]

I am going to talk about Clojure core.async. I really want to talk about the motivation behind it and how you should think about it, but I will talk enough about the details so you get a sense of what the API is like, but it is not fundamentally about how to use it, or the code.

[Time 0:00:20]

```
slide title: Problems

+ Function chains make poor machines
+ Real world concurrency is exposed via
  callback APIs
```

What problems are we trying to solve?

The first problem we are trying to solve is the fact that function chains make poor machines. And if you have heard me speak before about how objects are like little machines and they are not very good for doing logic, functions are like little units of logic that are bad for making machines. And it ends up that because we are moving to this world where people are trying to be more reactive, we have a ton of callback APIs on our hands, and we tend to connect to them with chains of function calls.

And we are in this situation where we actually need a program that is more like a machine, and we are trying to use tools that are for functional programming to do that, and it is a bad fit. So how do we fix this?

[Time 0:01:09]

```
slide title: Premise

+ There comes a time in all good programs
  when components or subsystems must
  stop communicating directly with one
  another.
+ Conveyance must become first-class
+ A reasonable program is organized around
  processes and queues
```

So the idea behind this – and this is something that I have said often – is that good programs should be made out of processes and queues. At a higher level, you want to move to queues.

And you want to move to queues in particular as soon as you are involved in any kind of conveyance within your application, because at that point, it is no longer nested logic. You are trying to move something from one place to another, maybe from some input, through processes in your system that perform calculations or transformations, and eventually out somewhere else. That is a one-way street.

So we want conveyance to become first class. We want to organize our programs this way.

[Time 0:01:51]

```
slide title: But

+ j.u.c queues only coordinate via thread
  control (blocking)
+ You must block a real thread on the end
+ No threads (and thus no queues) in JS
```

And we could always do this. If we are on the JVM, it already has queues.

Often people have said, "Why does not Clojure wrap queues?" And it is because there are already queues, and they are perfectly fine. It ends up they are not perfectly fine, and we are going to make them a little bit better with this, potentially. But they are there, and they work. But they do have some problems.

One is that the only way you can coordinate with a queue, with the java.util.concurrent queue – in other words to wait for data to be present – is to take an actual thread, a real thread, and block on the queue. And that has a cost we are going to talk about in a second.

Also, if we try to look at the whole scope of platforms that Clojure addresses, which also include JavaScript through ClojureScript, there are no real threads there, and there are no queues there of any real sort. So, using queues directly and looking for queues from the host is not necessarily something we can always do.

[Time 0:02:49]

```
slide title: Thread overheads

+ stack size
  + a problem when (very) many threads
  + N.B. this is not a scalability problem, it is
    an efficiency problem
+ wake-up time
```

Even when we can do it, there are overheads associated with threads. And people make a lot of this.

And, in particular, on the JVM, there is a definite overhead per thread, which is the stack size. Every thread has a pretty big stack, and if you tried to have hundreds or tens or hundreds of thousands of threads, you would consume a huge amount of memory. There is also wake up time and other overheads associated with threads.

People will often talk about that as if it was a scalability thing. Scalability is not about what happens inside one machine. It is about being able to add machines or add resources to make things scale.

But it is an efficiency problem. Are we making the best use of the machine by using a thread per connection? Oftentimes we are not, so we would like an alternative to that.

[Time 0:03:38]

```
slide title: Events/Callbacks

+ Listenable Futures/promises
+ a web of direct-connect relationships
+ difficult to reason about or control flow
+ _fragmented logic - callback hell_
+ handler list visibility, monitoring, control difficult
+ Observables/Rx etc only mitigate certain cases
```

So the API du jour is events and callbacks. This is all back, all the ick from a long time ago from UI development is now, like, the way to do everything. We have Listenable Futures and promises and callback handlers and async APIs for every kind of RPC.

And, invariably, they expose themselves as these Listenable future, or some sort of a callback. And what we ended up doing is we put some logic into each of these callbacks that says, on a button click or on a message coming over the pipe, or on whatever, do this stuff. And so the stuff is a little piece of logic that we hook up there.

And there are lots of these sources of events, and we end up with lots of pieces of logic. And we end up with this giant web of these direct call relationships, a lot like the kinds of webs we create in object-oriented programs.

And, similarly, they are very difficult to reason about or to control flow. And everybody understands this phrase, callback hell, and we are going to really look more carefully at what that means and what it is about.

But, fundamentally, I think it is about having to break your logic up into little pieces so that those pieces of logic can live inside handlers, when those pieces are part of a design, or a way of thinking about a state machine, or a way to approach the problem that was all of a piece. And the fact that you have divided it up has nothing to do with the way you want to think about the problem, and everything to do with the artifact of this mechanism for addressing it.

And it makes everything difficult. It is difficult to see inside these things, to see which callbacks call which handlers, to monitor them. On what thread are the callbacks going to be run, etc. etc. etc.?

And there have been various approaches to try to mitigate some of this with observables and Rx and things like that. But they only handle a very narrow set of cases, mostly filtering or making a stream like kind of approach to composable transformations on a single event chain. But if you really are trying to make a state machine that has multiple sources and sinks of events, you cannot just get it out of something like filter and map composition primitives.

[Time 0:05:55]

`slide title: Call Chains as Machines`

So what does that look like? The problem looks like this, essentially. I mean, I do not care if in and out are queues, or external API calls, or sockets, or button clicks on a browser, or whatever. It is just all sort of generically stuff comes from somewhere else. That is input. It is outside of your program, and somehow it is going to show up and say, "Wooh! Pay attention to me."

You have some logic associated with that, which you are going to put in a handler for that. And eventually that logic is going to have to produce something and move it along.

Okay, up above, the "in" to the logic and "out", those are all function calls, right? The input thing is: on something, call this function. And inside that function, you eventually call print, or send it out, or update the DOM, or some thing like that.

And this box is not a whole box because the way you thought about this job may have been a state machine. Maybe you are trying to implement some algorithm. You are trying to implement some sort of a state machine and, in fact, it involves multiple inputs and multiple outputs. But because you have to break it up into these pieces, you end up with these small, fragmented things.

And the problem is, if you really did have a state machine model, you had some logic appear that may or may not need to do something in particular, depending on something that happened in the logic down below. And how are you going to coordinate those two things, because they are both getting separate event streams, and they are both running on each event?

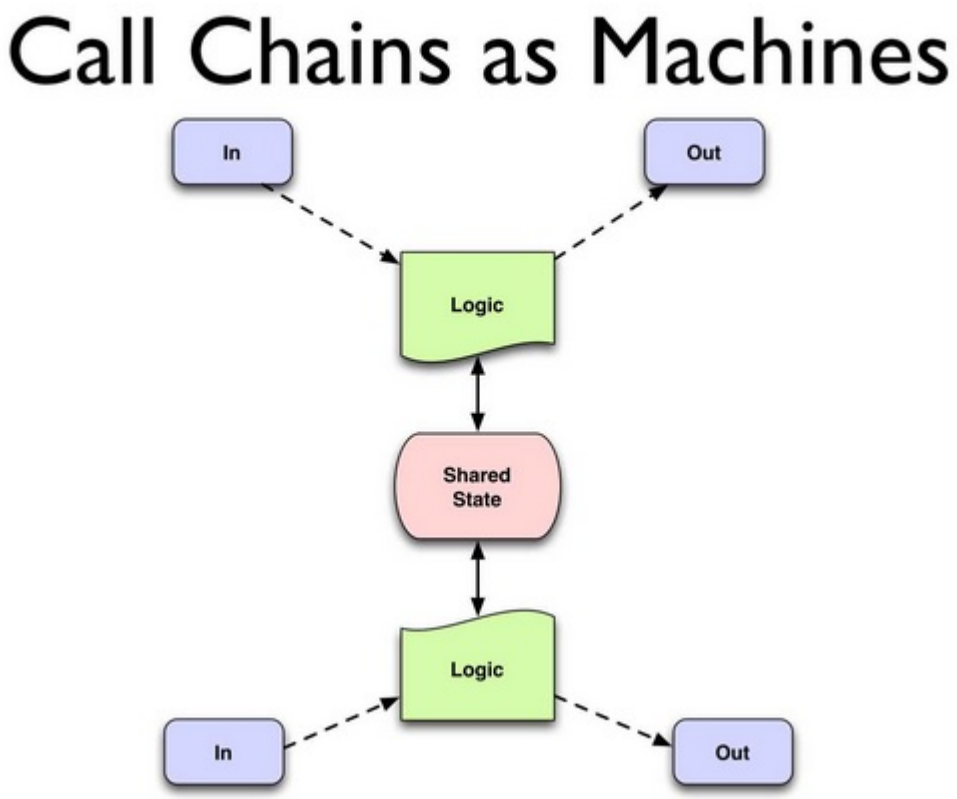

Figure 1: 00:05:55 Call Chains as Machines

And the way you do that, invariably, is having to introduce some shared state. And we all know the kind of party that leads to.

[Time 0:07:48]

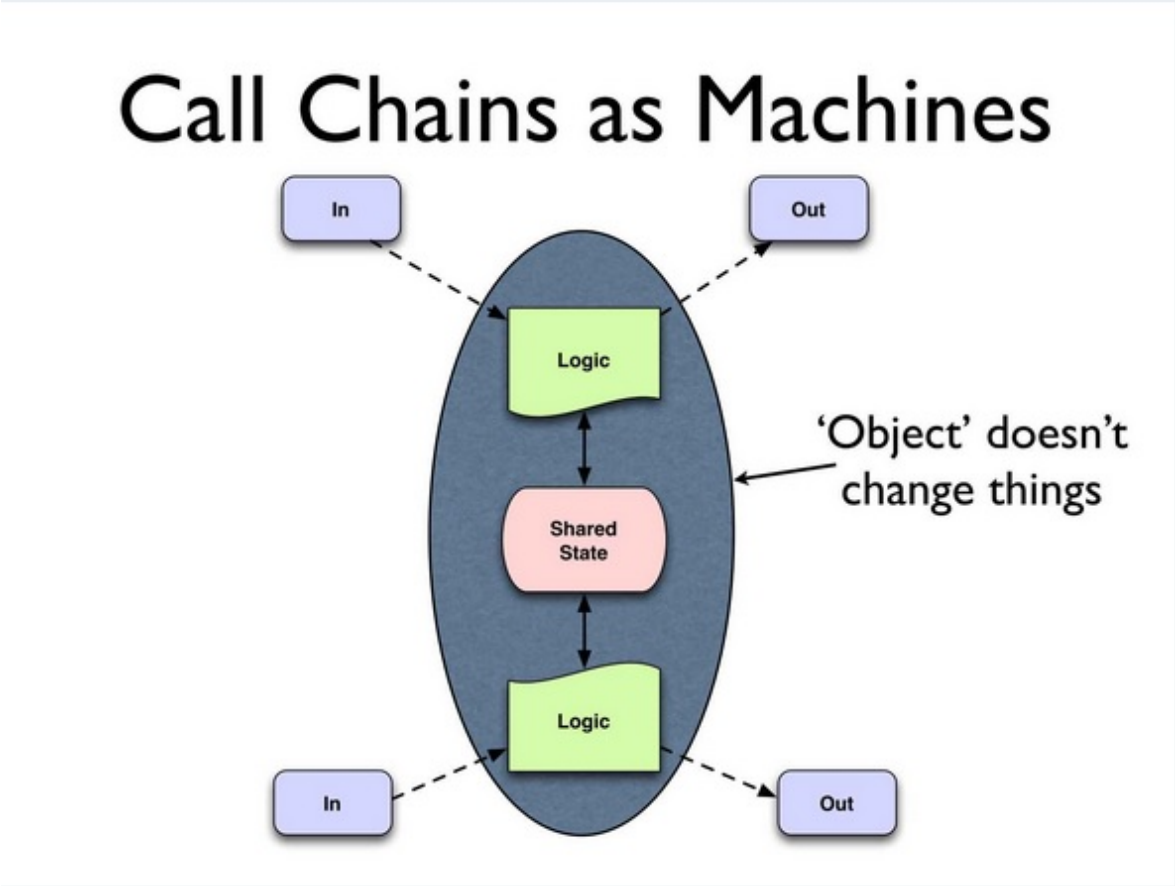

Figure 2: 00:07:48 Call Chains as Machines-build slide

And so objects do not fix this. Objects just put this in a blue oval. That is about –

[Laughter]

That is all that they do. There is nothing, nothing fundamental about what we just talked about is changed by putting it in an object. Okay? Objects are like marionettes, where anybody can pull any string at any time, and you are not going to get an episode of "The Thunderbirds" out of that.

[Laughter]

[Time 0:08:15]

```
slide title: C# Style Async

+ Attempts to re-invert control
+ Turns linear code into callback/state-
  machine for you
```

So there are interesting things out there. This happens all the time. Well, what is the problem?

The problem is we had to take what would have been a single thread of control, and we had to break it up into pieces so that we could run it on callback handlers.

And some people have done work in this area already. C# and F# have async primitives, not necessarily oriented around callbacks, but they do something; they *are* solving the problem of the, what I would call, reversion of control, which is that you want to call some asynchronous operation. You want to continue to use your thread or give your thread back to a pool and eventually, whenever you were waiting, whatever you were waiting for comes back, somehow resume control.

And so the way C# style async works is that it takes code that looks linear, but that calls asynchronous RPCs, for instance, and will turn the code in place into a state machine, and actually register that state machine on the callback handler. So your code looks ordinary and linear, and this inversion of control is happening under the hood.

[Time 0:09:20]

```
slide title: Copying C# Async

(let [fut (future ...)]
  (do-something-useful ...)
  (let [result (deref fut)]  ;; blocks thread
    (do-something result) ...))

(let [fut (future ...)]
  (do-something-useful ...)
  (on-complete fut   ;; callback handler
    (fn [result] (do-something result) ...))

(async
 (let [fut (future ...)]
   (do-something-useful ...)
   (let [result (await fut)]    ;; relinquishes thread
     (do-something result) ...)))
```

So it is an interesting idea to copy this. I do not know who saw Philipp's Scala Async talk earlier? Good. So it is not a bad idea to think about copying this. And Scala did that, and I saw Philipp's talk in the Spring, I guess. And I was like, hmm, that is not a bad idea. We could just copy that in Clojure, and it would look like something like this. And this is not what we ended up doing, but it shows you the idea.

So up top you have traditional blocking code. There is some future. The future is blocking. When you deref it, you are waiting. You are wasting, you are tying up the thread, and you are wasting resources.

In the second place, you use something like callbacks, so now you have a Listenable Future, and you would say: on completion of the future, you know, do this job.

And it looks kind of straightforward here, but if you were in the middle of a loop or something else, or you had multiple things you wanted to wait on, this would get really nasty trying to fabricate what the callback handler would be, and have a capture of the context of a linear looking program segment would be very difficult.

So what C# style async does is it says, well, just put that code in some sort of a special construct we will call "async". And then instead of registering a handler with the callback or actually blocking on it, you will make a call to await, which semantically *is* blocking. It is like the first code here.

But some magic – and in the case of C# is some compiler magic – is going to go and look at this block of code and say: Okay. I see that you are trying to do an asynchronous thing. Let me analyze this block of code. Let me turn it into a state machine that actually can be called back. Let me register that state machine as the callback handler for that future, and relinquish the thread. So the thread is no longer running.

And what happens is: the code that you want to have be the continuation, if you will, of that future is stored away in a blob of a state machine that will eventually be called upon to run on a thread from a thread pool. And that allows you to share, and it is really a great approach to this. It makes for linear code and allows you to use the machine efficiently because you can share this thread pool. You can have lots of these pending future actions.

But notice in particular this is really RPC, and that is sort of the problem.

[Time 0:11:56]

```
slide title: Just Sugar

+ Promises/futures are the one-night stands
  of architectural constructs

  _handoff or call/return_

+ Can't model enduring relationships
+ Not much help for external events
```

I walked away from the talk saying, "that would be a cool thing to copy". And then like after a weekend, I was like, "Uh." Because the promises and futures, it is not actually the first problem I talked about. The first problem I talked about was true asynchrony. Somebody clicks on a button or something comes over a socket. Your code did not initiate that the same way the code we saw on the previous slide did. That is code that says: Go and do this, and then when it is done, come back. And I want to relinquish the thread in the meantime.

These things are actually happening asynchronous, so promises and futures are sort of lightweight constructs, you know, that are one-night stands. They are just handoffs, or call and return scenarios. They cannot really model enduring connections.

And so they are not actually helpful for this. So it is sugar. I mean, it is good sugar, but I felt like we should put it on a better cake.

[Laughter]

[Time 0:12:49]

```
slide title: Queues/channels!

+ Decouple producers and consumers
+ First-class
+ Enduring
+ Monitorable
+ Multi-reader/writer

[ Photo of a conveyor belt with a line of small cubes on it. ]
```

So what do we do? Again, I think that people who are writing production server programs already use queues. This is the way most large server things are decomposed. They are not a giant, you know, web of direct calls. There are queues invariably inside everything.

And they have a lot of great properties. They decouple producers and consumers. Who is putting the things on one end of this? Who cares? Who is on the other end? Nobody cares. They do not know about each other. Somewhere down there someone is going to take this thing.

They are first class. There is that queue. There is that conveyor belt right there. You can go. You could kick it. Touch it. You can talk about it, name it, bump into it. It is outside.

They are enduring. The person who is on this end putting boxes on takes a break, and somebody else comes up, and the person on the other end is sick today, and somebody else comes out. But this conveyor belt is still there. It is there in spite of people coming and going.

Potentially, because it is first class and external and a thing, it is an easier place to hook on some monitoring. And it supports multiple readers and writers.

[Time 0:14:00]

```
slide title: CSP

+ Communicating Sequential Processes
  + Hoare 1978 (yes, more old stuff)
+ first class channels
+ blocking semantics by default
  + can be used for coordination
+ but can also be buffered, async
+ occam, Java CSP, Go
```

So, of course, there is nothing new to event. It is just a matter of finding stuff. So the thing to be found in this case is CSP, which goes back to Tony Hoare's work in the '70s. Sort of like the parallel track. There is actors and there is CSP. You know, you sort of pick your fight.

So we are going to take this side with core.async and bet on CSP, as many others have. And it has some very interesting characteristics. Maybe not always in the – not all in the first paper, but over time, CSP has come to represent first class channels as being the only way that independent processes interact with one another. There is no shared state except for these channels.

The semantics are blocking by default, which means that you can also use them for coordination, which is quite interesting. So it is not just a primitive for conveyance. It can be a primitive that allows you to say, "stay here until somebody is ready for me", or vice versa.

They can also be buffered and, therefore, allow for some asynchrony between the producers and consumers. And there is a long set of history behind this: Occam, and implementations for Java. And, of course, the most recent rendition that sort of makes this first class and a central point of the language is the Go language. So we owe everything about what you are going to see to all of this prior work.

[Time 0:15:24]

```
slide title: More cool stuff

+ Multi-reader, multi-writer
+ pass endpoints around
+ select/alt
```

```
  + wait for any one of a set of IO ops
  + including writes, and timer
+ Algebra, formalisms etc if you're into that
```

But there are lots of cool attributes to this. The multi-reader, multi-writer part is important. You can pass endpoints around as first class things.

The other cool thing about CSP and channels is that they support choice via a mechanism called either "select" or "alt" depending on what language or what paper you are reading. And the purpose there is that you can wait on one or more of a set of IO operations on channels. Those include both writes, reads, and timeouts.

In addition, through time and in the academic literature and work, there is formalisms improved some things like that that can be used to help you reason formally about systems built in this style. There is nothing in that area yet done for core.async.

[Time 0:16:13]

```
slide title: JVM Impls Exist

+ Java CSP
+ Communicating Scala Objects
+ both tied to real threads
```

So there are already people that have done this. There is a very nice implementation of CSP for Java called JavaCSP. People have tried stuff in Scala. But both of these implementations are tied to real threads. They basically use real threads as processes, one-to-one.

And we want to do something else for Clojure.

[Time 0:16:32]

```
slide title: The Challenge

+ Create a channels API for Cloure _and
  ClojureScript_
+ Similar calls support both
  + real threads, real blocking
  + macro-generated inversion of control
    (IOC) (thread relinquishing, thread pools,
    callbacks)
```

In particular, we want to try to create an implementation of this approach that works both for Clojure and ClojureScript. And that supports both the problem of: I want to use real threads and real blocking, which *still* has utility if you do not have an arbitrary number of connections you need to support. If you have a small server with a finite number of processes, it can be more efficient and have higher throughput to use genuine threads and real blocking than to force everything to go through the mechanism of a thread pool.

But if you are trying to target arbitrary connection counts, or work on a platform that does not have threads at all, which would be the JavaScript engine, you need something else.

And so the idea here, if there is any new idea in core.async, it is just to implement this technology on platforms that were not designed for it. Like if you look at Go, it has a runtime engine that is oriented

around this, and to get the more efficient use of threads and thread pools by leveraging the kind of inversion of control technology that was used in C# async. Because that is what you need to take code that should logically be blocking, and instead turn it into data and eventually consume a thread from a thread pool. So that is the idea.

[Time 0:17:54]

```
slide title: The Opportunity

+ Support reasonable model across spectrum
  + traditional threaded apps
  + high connection count servers
  + evented servers (node, vert.x)
  + the @&#%! browser already
+ and beyond?
  + network channels, etc.
```

So if we can do this, this is really great, because we can deal with traditional threaded apps with low thread counts. We can approach designs for high connection count servers that are efficient. You can deal with evented server APIs like those provided by Node and Vert. And we can finally try to make some sanity in browser development.

Not targeted currently by core.async is anything to do with extending this over the network.

[Time 0:18:24]

```
slide title: core.async

+ A library
+ Independent 'threads' of activity
  + real threads and IOC
+ queue-like _channels_
+ Clojure on the JVM and ClojureScript on JS
```

All right, so core.async itself, what is it? It is a library. It is about fundamentally coordinating between processes using channels. But the processes you can think of as independent threads of activity.

They may be real threads, or they may be these inversion of control threads, where you write code and it looks like it is doing X, do Y, do Z in a row and blocking in the middle. And that process is turned into data whenever the blocking occurs, and the real thread is relinquished. So we will call those inversion of control threads and, "threads" like this.

But logically they are threads, and the semantics are the same. We want the semantics of threads and the semantics of *blocking* in all cases, because it allows us to write linear code. And then we will connect these things with these queue-like channels and target both platforms.

[Time 0:19:13]

```
slide title: Threads and Blocking

+ (thread body...)
  + real thread, real blocking
+ (go body...)
  + IOC 'thread', state-machine, 'parking'
```

So the first thing you need to be able to do is create a thread or create a process, and there are two ways to do it. There is the thread call, which is a lot like the future call, except this could tap into different thread pools. And is clear about what you are doing. When you say thread, you get an actual real thread.

And when the use the APIs within this thread, you get real blocking. So it is not quite as – you cannot create as many of these arbitrarily as you could with the next ones, but when you want real threads and you know what you are doing, you can get high throughput with something like this.

And then we have Go. And Go is the same kind of thing. It creates a logical thread. It runs the block inside of it in this mode whereby if you issue any blocking calls, this mechanism will come into play, create an inversion of control state machine out of your code, and register it on the handler for the asynchronous callback.

So we are going to talk about that as a thread, where blocking we are going to call "parking", which essentially relinquishes the thread and takes the code and parks it somewhere for its resumption later.

[Time 0:20:27]

```
slide title: Channels

+ Queue-like
+ Multi-writer, multi-reader
+ _Blocking_
+ Unbuffered, fixed buffers
```

All right, so channels themselves, what are they? They are like queues. A queue, you put something on one end; you take stuff off the other end. That is it. That is what they are about. Put stuff in one end; take stuff out the other end. You can have multiple writers. You can have multiple readers.

Fundamentally, channels are blocking, like if you do not touch them or do anything else, they are blocking. That means that if somebody comes into read and no one has written anything, it waits. If somebody comes in to write and no one is waiting to read, the writer waits, so they block on both sides. By default, they are unbuffered. We also support some fixed buffering, so I will talk about that in a second.

[Time 0:21:01]

```
slide title: Channel API

+ Create: (chan), (chan n), (chan buffer)
+ Put: (>! ch val)  ; 'parking'
  (>!! ch val)      ; blocking, not on JS
+ Take: (<! ch), (<!! ch)
+ Close: (close! ch)
+ Can mix modes (!,!!)
```

So the API is pretty straightforward. You create a channel with chan. Chan, with no arguments, creates an unbuffered channel, which is essentially synchronous rendezvous. You can say chan with N, which gives you a buffer of that size, or chan with a buffer. And there are a couple of API calls that allow you to create buffers.

Then we can put with – put, that is how we read that: right arrow, bang (>!); right angle, bang (>!); and that is the parking variant. All right. That is the one that will park the thread. It must be used within a Go block.

The other variant is the double bang (!!), and you can read those as "blocking", put blocking. And all those flavors, and this will go throughout the API. If there is a double bang (!!), it is a blocking call. You read it as blocking, so put blocking. And *none* of those are available on ClojureScript, on JavaScript, because there are no threads to block.

Similarly, there is take, and take blocking, and close. What is really useful about this, if you are using it on the JVM, is that you can mix and match these, so you can have a bunch of Go threads, and you can have actual real threads. You can make blocking calls and parking calls on the same channels on different ends of the same channels. It all fully interoperates, which can be extremely useful.

[Time 0:22:15]

```
slide title: Buffers

+ Unbuffered == rendezvoud
+ Fixed will block puts when full
+ Fixed variants (never block):
  (sliding-buffer), (dropping-buffer)
+ Unbounded == _bad_
```

So the buffers themselves, again, normally we are unbuffered or, by default, we are unbuffered. And that acts as a rendezvous, like I said. Readers wait for writers and vice versa.

If you supply a fixed buffer, you use N of some value, then writes will succeed right away until the buffer is full, and then the writes will block. You know, "block" like this.

And then there are two flavors of buffers that will never block a writer because they have incorporated a policy about what to do when they are full. And one is a sliding buffer, which effectively walks across and drops the newest stuff, and a dropping buffer, which is, if you come to it with new stuff and it is full already, it just drops whatever you send. And they will never block the provider.

[He said "drops the newest stuff" above, but the documentation string for sliding-buffer says "oldest elements in buffer will be dropped", so I think he meant to say "drops the oldest stuff".]

What we do not provide, and *will not* provide, are unbounded buffers. This is just a recipe for a broken program. It is just like, "I do not feel like seeing this bug until later. So I will make an unbounded buffer."

[Laughter]

I am not going to help you do that. We have already probably rejected the patch request a couple of times, and we will keep doing it. I mean, just, hopefully people will get tired of submitting it.

[Laughter]

Just don't do it. I mean, what is beautiful about this is that you can establish a policy. Eventually we will have a recipe for providing other policies with more sophistication, but make a decision. You *need* to make a decision here. You cannot just have stuff piling up all around in memory and not be thinking about it. So it is a value proposition of CSP and of core.async that you are thinking about these things and making choices.

[Time 0:24:00]

```
slide title: Choice

+ Often useful to wait on multiple ops
+ alt(*) can block on puts/takes+
```

```
+ (alts! ops), (alts!! ops)
  op = take-ch or [put-ch val]
  functions
+ If more than one ready - random, or in order
+ _One and only one op will complete_
```

So the other cool thing, extremely cool thing about CSP and about channels is the fact that they offer choice.

So it is quite frequent that you want to make, you want to put a state machine in particular in a state where it is waiting for one or more possible activities or operations to succeed. And that can be a really difficult thing to do. That is something, for instance, that none of the Java queue primitives support at all. Windows has some nice primitives for wait on multiple things, but Java does not. Obviously sockets do have select and whatever.

So alt and the two flavors of alts are functions that take a set of operations. An operation is one of two things. If it is just a channel, it means try to read something from this channel. If it is a vector with a channel and a value, it means try to put this on a channel. And alts will return whenever one of those things becomes ready to complete.

By default, it will be random. There is also a priority option that will let you say try them in this order. What is really important about the choices that one and only one operation will complete, so you are going to ask for, let me know when any one of these things has happened, but only one of them will happen. It will be as if all the other pending requests that you made were cancelled, and you just got the one thing.

And you get a return value, which indicates, in the case of take, it is just the value. And in the case of take where you had multiple channels, it is going to tell you which channel actually succeeded and what the value was. In the cases of put, it is just going to be like, okay, that worked. That channel put worked.

[Time 0:25:42]

```
slide title: alt!/!!

+ Macros, combine alts + cond
+ Matching ops returns [val ch], _binding_

     (alt!
       [c t]       ([val ch] (foo ch val))
       x           ([v] v)
       [[out val]] :wrote
       :default    42)
```

And then there is a macro that combines this function with cond, with the logical branching. So basically you say alt, bang (alt!), which is not a function now. This is a macro. And you supply one or more sets of operations you want to attempt. The whole set of operations is going to be tried sort of in parallel, and the first thing that is available to complete will be the result of the condition. So we have – I think I have arrows here. Let us see.

[Time 0:26:12]

```
slide title: alt!/!!
```

```
+ Macros, combine alts + cond
+ Matching ops returns [val ch], _binding_

             (alt!
        ---->  [c t]       ([val ch] (foo ch val))
 op(s)  ---->  x           ([v] v)
        ---->  [[out val]] :wrote
               :default    42)
```

So we have the operations themselves. The first two are reads, try to read from C or T, or try to read from X, or try to put this val to out. Or if none of them are ready right now, just return 42.

[Time 0:26:27]

```
slide title: alt!/!!

+ Macros, combine alts + cond
+ Matching ops returns [val ch], _binding_
                                  /
                                 /
                                /
             (alt!              V
        ---->  [c t]       ([val ch] (foo ch val))
 op(s)  ---->  x           ([v] v)
        ---->  [[out val]] :wrote
               :default    42)
```

Then we have expressions, which would be the result. If C or T returns a value, then we want to grab that value and call it val and grab the channel that actually succeeded because it could be C or T, and call that CH. And then we will call foo with those two things. So we have a binding and then some expressions.

[Time 0:26:45]

```
slide title: alt!/!!

+ Macros, combine alts + cond
+ Matching ops returns [val ch], _binding_
                                  /
                                 /
                                /
             (alt!              V
        ---->  [c t]       ([val ch] (foo ch val))
 op(s)  ---->  x           ([v] v) <-----    ^
        ---->  [[out val]] :wrote <------\   |
               :default    42)            --exprs
```

So this is a macro that is an expression. It has not got all the gook of Go with the statements. And it will return one of the values.

So you are going to try multiple operations, and you can have a default if none of them are ready. Otherwise, you will block waiting for one of them to succeed. You can get the value that you desire as a result of that particular operation succeeding. So it is like alts plus cond.

[Time 0:27:14]

```
slide title: Timeouts

+ Just channels
+ Create: (timeout msecs)
+ Returns channel that closes after msecs
+ Include take of timeout in alt(*)
+ Timeout channels can be shared
```

So how do you do timeouts? Because a lot of times what happens is you are setting off all these blocking things, but you do not want to wait forever. What is really cool about Go is that they decided to make timeouts channels themselves, and that is a really good idea, so well worth copying.

So you just create a timeout by calling timeout with the number of milliseconds. You get a channel that closes after that number of milliseconds. So you can put that into an alts, and what will happen is either you are going to get something from the thing that is not a timeout, or the timeout will close, and that will cause your alts to have a completed operation, which is the closing of the timer channel.

But the cool thing about it is: that timeout now is a real thing, as opposed to how many people like putting timeouts in API calls? Yeah, that stinks, right? Especially if you tried to coordinate multiple things or you are just not sure how long to wait, or you were trying to do stuff in a loop where you have to keep trying things, but you are trying to have an overall counter go down, so you have to keep recalculating that timeout to smaller, smaller, smaller, smaller.

And now you can just create a timeout once and say: I am going to try all this stuff in a loop for three seconds. You create a three-second timeout, and you keep passing that same thing into every API call, the same one, because it is either going to complete or not. And you can just reuse it, so you can share them, which is very powerful.

[Time 0:28:28]

```
slide title: Like Go, but ...

+ All of the same operations are _expressions_ (not
  statements)
+ This is a library, not language feature
+ alts! is a _function_
  (and supports a runtime-variable # of ops)
+ Priority is supported in alt
```

So obviously if you know, Go, this is very similar. It is like Go, but the things that are different are: all of the operations, everything I have shown you, they are all expressions. It is for a functional programming language. There are no statements. You do not need to have state to start interoperating with channels, because channels are a state mechanism already.

It is a library. It is not a language feature. It uses macros to do what it does. And the macros are quite interesting, and we will have to have other talks about those. But the Go macro is the thing that inverts the control and sets up the state machines for you.

The other thing that is interesting is that alts is a function. You can map it. It is variadic. It supports a run time variable number of operations. You cannot say that with any construct in Go because they are all statements. And we support priority.

[Time 0:29:15]

```
slide title: At the Edges

+ Callbacks exist already
  APIs etc
+ Lost cause?
  No - enqueue on channel in handler
+ Use (put! ch val)
  + Need not be in (go ...)
+ Similar (take! ch cb)
```

So how do we integrate this into our world, because we are not going to get everybody to change their APIs to use channels. They already use callbacks. Are we doomed?

And the answer is no. The trick is, as soon as you are in a handler and something comes in the handler, just put it in a channel and that is it. You should be done. Do not put anything more there.

So we actually have, because the operator is the put, like the right angle (>), yeah, the right angle, bang (>!), put can only be used inside Go, and you are not in a Go block in this event handler. Right? This is, like, on a mouse click. That was not in a Go block.

What do you do? We have two calls, put! and take!, which can be used from outside of Go blocks that are asynchronous and will just enqueue the request and return right away. And it is sort of an entry point from the edges of your system into the channel based system. So you are talking about a process that was not really a CSP process has got some input. You use put! to introduce it into this system.

And there is a similar take!, which reverts control. This is the way to get out, for instance in JavaScript, and get a callback back.

[Time 0:30:32]

```
slide title: The Final Frontier

+ The browser
  all callbacks, all the time
+ _Friends don't let friends put logic in handlers_
+ ClojureScript + core.async restores separation
  Events, logic, view
+ Clean, flexible routing
```

So speaking of JavaScript, so what do we do there? This is the browser. It is just *all* callbacks. It is 100% callbacks.

And again, this is totally fine. Everything I am showing you – everything about the Go blocks, not the thread ones – it all works in the browser, all works in ClojureScript. If you read David Nolen's posts about this, he is doing fantastic things that are exactly what this was designed to do.

You just revert control immediately in your handler. *Don't* put any logic in your handlers. Just take the event, turn it into data, put it on a channel, and be done with it. And that will let you put your world right side up.

So it is just a way to restore the separation of concerns. Instead of fragmenting your logic and building all these little pieces everywhere, you keep your logic together. You use these bridges from channels to get stuff to route through.

[Time 0:31:28]

```
slide title: Model
```

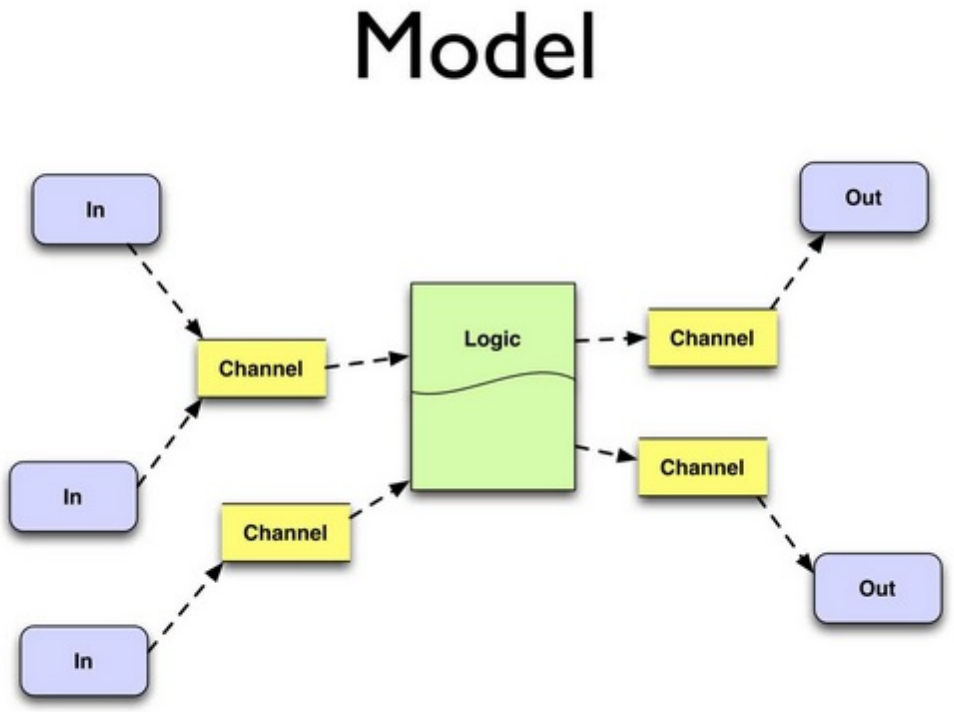

Figure 3: 00:31:28 Model

So the model essentially looks like this, in contrast to the first one. You are going to just take inputs and put them on channels. You are going to have your logic all together. It can consume from multiple channels. It can put on multiple channels. It can coordinate with other logic and other instances of the logic, because it is many to many. But it is all together, and it puts it on channels. And eventually those reach the outside world.

[Time 0:31:55]

And so the super critical thing is this part of the talk, because we have two solutions here that, on the tin, they are the same. These let you efficiently do asynchronous programming. It is like, great. It is a feature. Awesome. Dude, let us go do it.

But they are really, really different. The characteristics are completely different, so it is a battle.

Direct versus indirect, right? So how many differences can you see between these things, except for some colors? It looks like more of a hassle, right? [indirect] I have five things down here. [direct] I have four things up there. Always a bad measure.

[direct] So what happens to your logic in the direct system? It is split up into tiny pieces and spread out through callback handlers. [indirect] In an indirect system it is coherent. It is all together. It is linear.

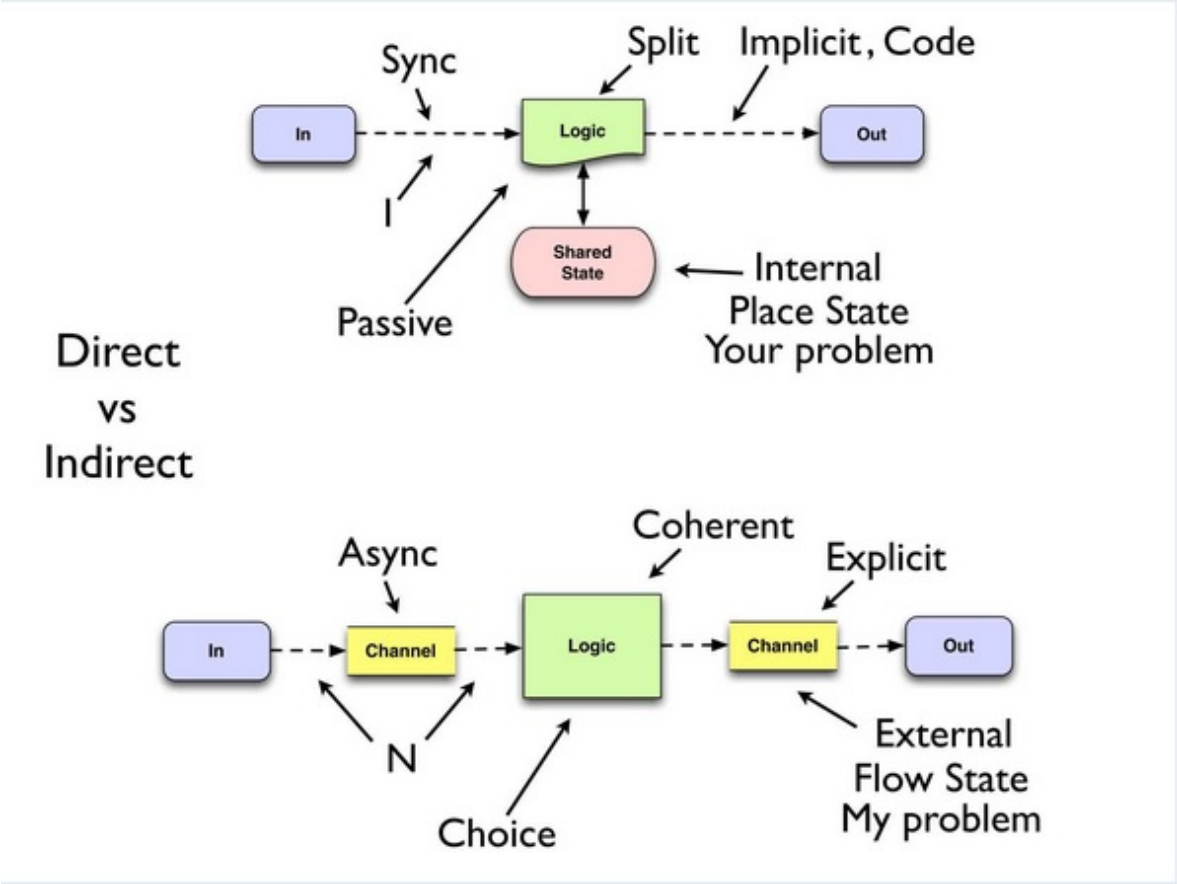

Figure 4: 00:31:55 Direct vs Indirect

[direct] What happens on your callback? It directly calls your logic, which directly calls the output. It is all this big chain of fire, fire, fire, unless you manually introduce some more asynchrony. It is synchronous.

[indirect] Channels can be synchronous or not. We can set them up with no buffering and actually cause workflow and backpressure waiting, or we can add buffering and get some more asynchrony there. We still have choices in the logic itself.

[Time 0:33:19]

[direct] What is the arity of the callback to the callee? It is built in. It is like you gave them a piece of code to call. They call that code. It is one-to-one. A button click calls the thing. On socket calls the whatever.

[indirect] It is N-to-N here, with asynchrony. How many people can feed this channel? As many as you want. How many people can consume it? As many as you want. So you can share the work.

You can have multiple sources. You can do routing. You can create these relationships dynamically. You can pass the channel to talk to dynamically.

[direct] This other stuff has got code. I made code, and the code has got like what to call up above, right? So that is all implicit. Where am I going from here? It is like inside this piece of code.

[indirect] Where am I going from here? I do not know. I mean there is going to be some code that wires this stuff together, which I am going to be able to see and think about, and cooperate with, and possibly manage dynamically, something you cannot possibly do in the other scenario.

[direct] Where is the shared state? Well, sort of an internal implementation detail of whatever you had to do to try to make your state machine cross multiple callback handlers, which is usually just a real incredible mess.

[indirect] Where is it here? It is external. It is not like there is no state. There is state. Of course there is state. It is a big machine. It moves stuff from there out the other side.

[direct] But in the top we are making function calls. But how do we know we are using function calls wrong up top?

[Inaudible comment from the audience.]

[Time 0:34:45]

Right, we – functions calling functions calling functions. There is nothing wrong. That is a call chain. We do that all the time in functional programming.

But what else do we also do? We pay attention to the return values. Function calling function calling function, getting the return, using it, getting the return, using it, getting the return, using it. It is fine. That is function composition to have a stack of functions serve an algorithmic purpose.

That is not what we are doing up top. We are shoveling stuff across functions. We are ignoring the return values, and trying to spew stuff out into the real world out the other end.

[indirect] So having the state be external is totally fine. It is now clear that there is state.

And I think this is very, very interesting. These states are different.

[direct] The shared state that you have between two pieces of logic in order for them to communicate with each other is going to *place* state. It is going to be, "I put something in there. I squirreled away some acorn so that either when I come back or if this other handler comes in, they know that I am in the middle of doing this, and they should not do that." Or we have got six of these, and we need four more before we can proceed. There is going to be the shared state, which is a set of places that you are using to update things.

[indirect] But I think what is really cool about first class channels is that they expose a subset of state, and it is a very interesting and far safer one, which I will call "flow state", which is just about: all you can do with this kind of state is put stuff in it, and then you really do not know anything more about it. Or you can take stuff out of it, and you really do not know how it got there.

And the analogy I would make is, let us say you work at a factory. And you get into the factory, and you have your jacket. And you are going to do something with your jacket. Would you rather hang it on a coat hook, or put it on a conveyor belt that is moving, or dump it down a laundry chute, or put it in the back of the UPS truck that is about to leave?

[Time 0:36:44]

There is a difference, right? Those are all places to put your jacket. But what is it about the coat hook? You expect to go back there at the end of the day and see your jacket still there. Of course, if there are only a few coat hooks and there are a lot of jackets, you have this contention problem because it is a place. But you would never put your jacket on any of those other kinds of places, states, because you know they are flow.

They are going to go away. It is going to go on a conveyor belt. It is going to go somewhere else. It is going to go down that chute and you cannot recover it. The UPS truck is going to drive away.

And that is good, because you cannot possibly have your life depend on going back and seeing the same thing there again. So it is not really like state the way shared state and place oriented state is. Flow state is much simpler. It is much easier to reason about and much safer, because you do not have this expectation of coming back.

And then in any case, if you are running any of these things in an environment in which not only is there cooperative concurrency, but there is actual, real, simultaneous concurrency, then any kind of state, whether it is a channel or the shared state there is going to have to have some coordination, so we do not all step on each other.

[direct] But in the first case, whose problem is it? It is your problem, that coordination. Getting that thread-safe is *your* problem.

[Time 0:38:10]

[indirect] In this problem, in the bottom case, it is *my* problem. It is the core.async library problem. And once we get it correct, I think we have already gotten it correct, but it is correct for everybody. Its correctness is shared by everyone. So which do you want?

[direct] Also, the logic: When do you decide to handle an event that comes through a callback handler? Pff! You don't decide. You are, like, you are just passive. You get called whenever you get called.

[indirect] When do you have it when you have channels and alt? Whenever you choose. If you do not want to go read that channel right now, don't do it. If you think it is more important to read this other channel, or to write to these channels, or to wait for ten minutes, or to go ask somebody, you are not a passive recipient of control being driven from an external event, unless you have chosen to do that.

So I think the problem I labeled first is the one I want to call out now. This first thing is using code as a queue. Call to call to call to call. When we should always be using data. Again, it is one of these things. We never make this mistake when we go over wires. We never make this mistake. When we go over wires, we don't. The reason is because we cannot.

Wires do not let us make function calls. We always have to turn it into data, put it in a buffer, enqueue it, shovel it over, dequeue it, do whatever. Maybe we will map it again to a function call and we will recreate RPC, but there is no RPC. Wires do not do RPC. They are not code. They do not have entry points. You cannot invoke them.

[Time 0:39:56]

So we never do this. But it is a good example of another thing that when we bring it inside, we make this mistake. And when we are shoveling stuff around inside our programs, it is not different than when we are shoveling around between boxes. It is not different. It should not be a different way to think about it.

[Time 0:40:14]

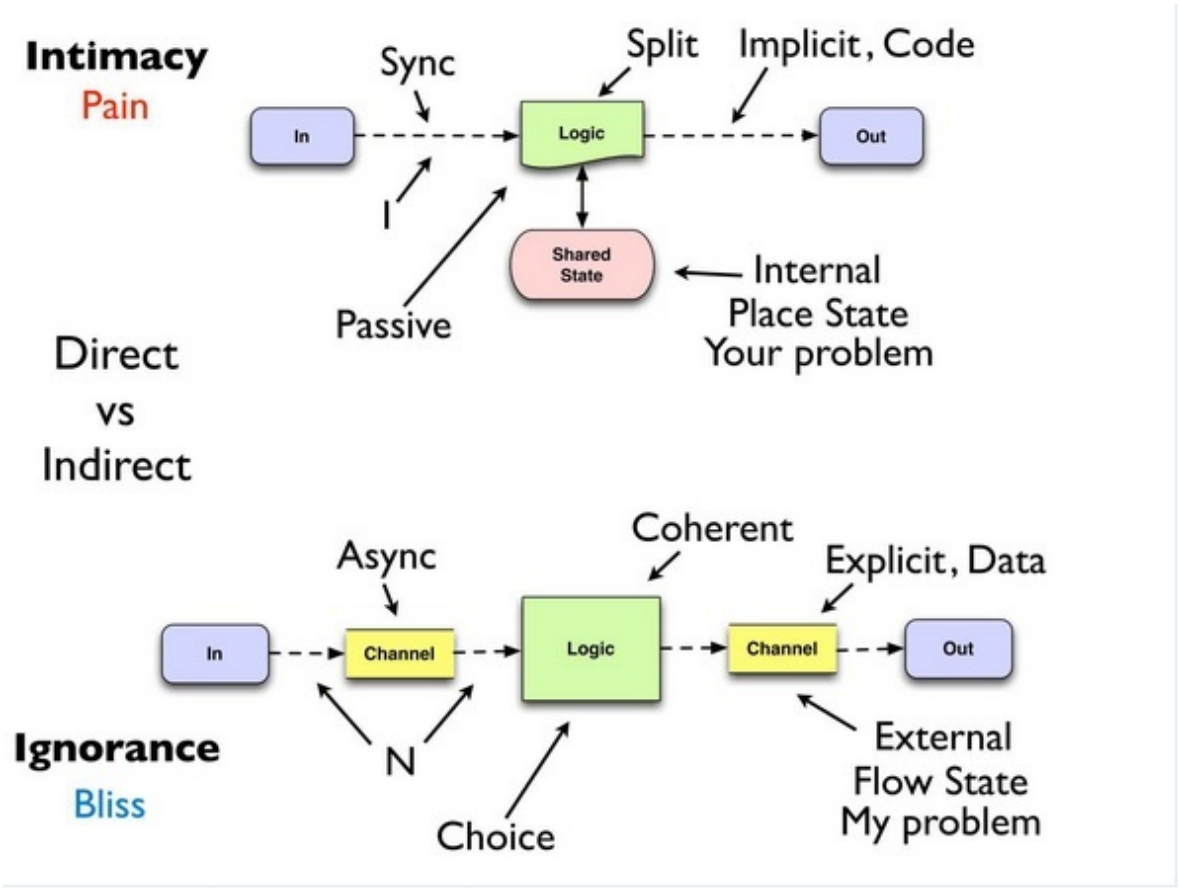

Figure 5: 00:40:14 Intimacy Ignorance

So there is a sense in which the direct callback approach is intimacy. It is making pieces aware of the other pieces. The callback event knows who to call. The caller knows where it goes next, and it is all code.

And there is a sense in which using channels and this indirection is ignorance. And we all know that ignorance is bliss.

[Laughter]

And it ends up that, in systems, it is also bliss.

[Laughter]

At least in systems . . .

[Laughter]

intimacy is pain. This is where the pain comes from. This is why. This is why it is painful.

[Time 0:40:54]

```
slide title: Code

(defn search [query]
  (let [c (chan)
        t (timeout 80)]
    (go (>! c (<! (fastest query web1 web2))))
    (go (>! c (<! (fastest query image1 image2))))
    (go (>! c (<! (fastest query video1 video2))))
    (go (loop [i 0
               ret []]
          (if (= i 3)
        ret
        (recur (inc i)
               (conj ret (alt! [c t] ([v] v)))))))))
```

https://talks.golang.org/2012/concurrency.slide#50

So here is a little bit of code, just so that I do not just show ideas on the screen and get complained about. So code. This is a great example from Rob Pike's examples for Go, and it is just nice because it shows everything.

I am not showing the supportive code, but the basic idea here is that you have got a job. You are trying to do this search. You need to come up with a web, an image, and a video result for it. You have a couple of possible sources of these answers. They may reply at different speeds. You want to bound the amount of time that you are going to take to do this entire job to 80 milliseconds. You want to try to get everybody to do the job, and take the first results you get, and get out.

So the function "fastest" takes a query and two sources, two URI endpoints that it can call. It is going to create a channel. It is going to set off Go processes to go and attempt to do that RPC with each of those web servers and get an answer. And each of those callbacks are going to go and take the answer they get, and put it on that channel. And then "fastest" is going to return the channel. So essentially you can read this as: set off two processes racing to put an answer on a channel and return the channel.

And you do that similarly for the images, and similarly for the videos. So we end up with three channel results. And we are going to set off asynchronous processes that are going to go and say: Read from that channel, then put it on this shared channel. So, and it is going to do one. So that read and then put back, that is the RPC kind of thing, and there is no – you can build that C# style async RPC stuff out of channel read and channel write like this, and it is a common idiom.

So it is going to read each one and put it on there. And then we have a loop. And the cool thing is that this loop, it does not know where this stuff comes from. It is just been told there are going to be three answers on this channel. Do not spend more than 80 milliseconds trying to read them.

And that is what it does. It goes through it. It makes an alt call, and it says: Try to read from the channel, or time out. Take the value and just add it to the vector. So this vector is going to return up to possibly three results, or nils if no results are available. But it is going to be done in 80 milliseconds no matter what. Given that spec for "this is the job you want to do", that will be really hard to write without stuff like this.

[Time 0:43:22]

```
slide title: What You Get

+ Separation of concerns
+ Coherent, linear logic
+ Recursion vs mutation
+ Coordination, backpressure
+ Dynamic configuration
+ Efficiency
```

So what do you get from using this kind of technology? You get a separation of concerns. You get it back, because you have lost it. You get coherent and linear logic. You move away from mutation. If you have a state machine inside your process, you can implement that using recursion instead of having to create internal state with traditional measures.

You get coordination possible. You can get backpressure with this, which is quite useful and difficult to get otherwise. You can dynamically reconfigure these networks, and you can use your thread pools and your thread resources efficiently.

[Time 0:43:57]

```
slide title: Thanks, Team!

+ Timothy Baldridge
+ Ghadi Shayban
+ Alex Miller
+ David Nolen
+ Alex Redington
+ Sam Umbach
```

So I would just like to thank the guys that helped me work on it, and particularly Timothy Baldridge did the Go macro inversion of control stuff, and it is really cool. And when he talks about it at some conference in the future, make sure you catch it.

[Time 0:44:10]

```
slide title: Try it

+ Code: https://github.com/clojure/core.async
+ Docs: http://clojure.github.io/core.async/
+ Blog post
  http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html
```

So this is where you can get the code and try it out. It is on GitHub. There are docs there. There is now a Maven artifact for it, and there is a blog post that describes it more, but that is it. Thanks.

[Applause]

[Time 0:44:21]