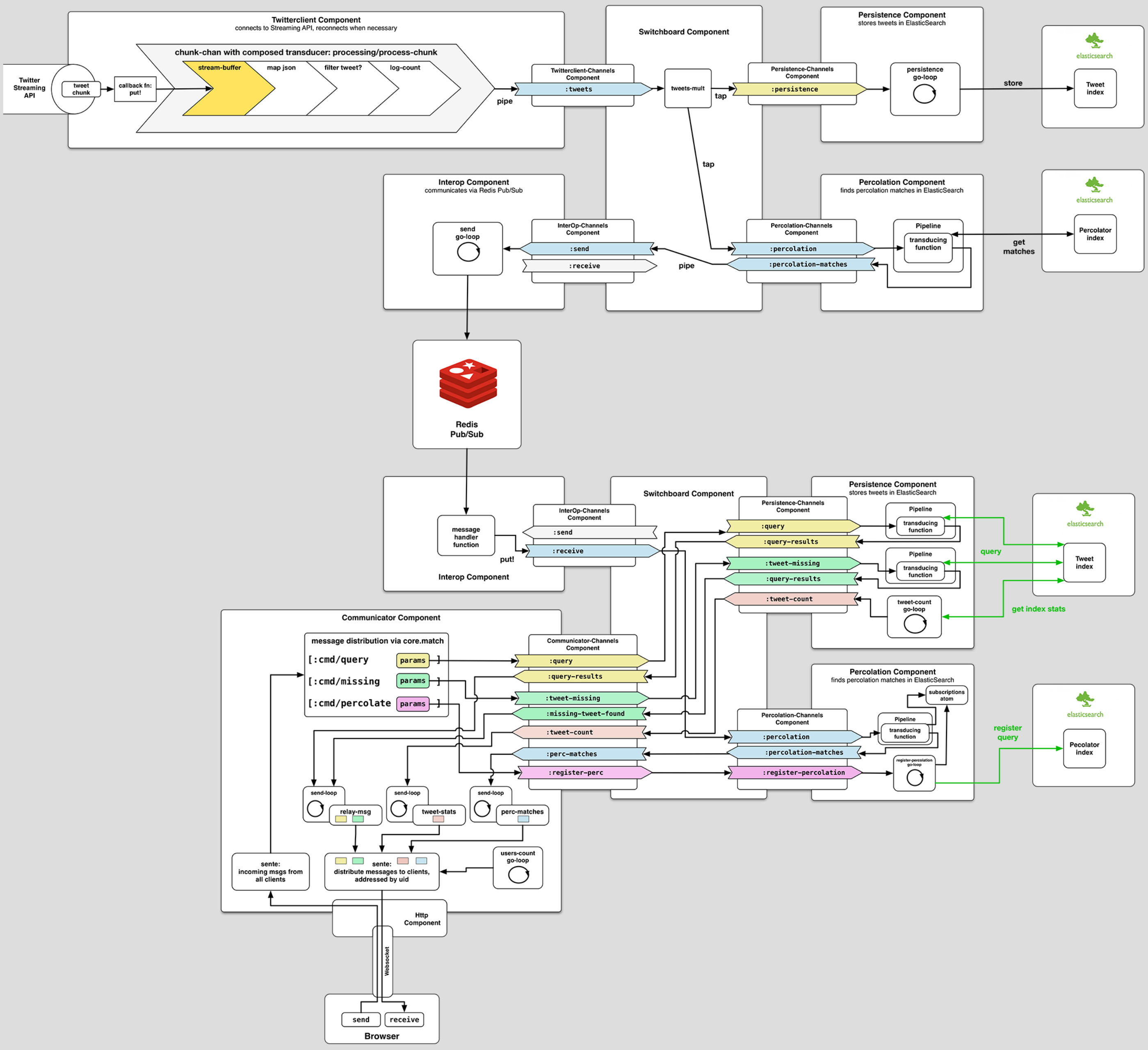
Building a System in Clojure and ClojureScript

Matthias Nehlsen



Building a System in Clojure (and ClojureScript)

Matthias Nehlsen

This book is for sale at http://leanpub.com/building-a-system-in-clojure

This version was published on 2016-03-10

Contents

1. How to use this Book – Update February 2016

It’s been a while that I was last able to put some creative thought into this book, and there are some reasons for that. Probably the most important one being the lack of a “traditional” publisher. When I first heard about LeanPub, I thought, what a great idea to let the readers provide feedback and thus replace the editor. But I haven’t found that to work very well, especially not while doing rather demanding consulting work at the same time. It seems to be an unfortunate reality of life that usually the loudest voice gets heard and served first, and with little feedback from you, dear readers, I didn’t get around to doing much on this front at all.

So please, if you want new content more often, please do provide some feedback and don’t think, hey, someone else will do it. Also, while occasional praise may be kind, I’m more looking for constructive criticism. It’s sweet that I have received some positive feedback, but of the aforementioned constructive criticism, I so far have received exactly zero. So, please, save time on messages that are only compliments, they aren’t actionable in any way and instead tell me where there’s room for improvement. I’ll also be happy to set up a call with you if you need clarifications on any topic. Please send me an email to matthias.nehlsen@gmail.com and we can take it from there. Even if you have nothing particular to say other than that you’re reading the book, I will still be happy to hear from you.

As an additional incentive for getting in touch with me, I will write little teasers for future chapters. When you find any of those particularly interesting, you can get that chapter earlier by letting me know. I **need** and appreciate your feedback.

That’s all.

Have fun.

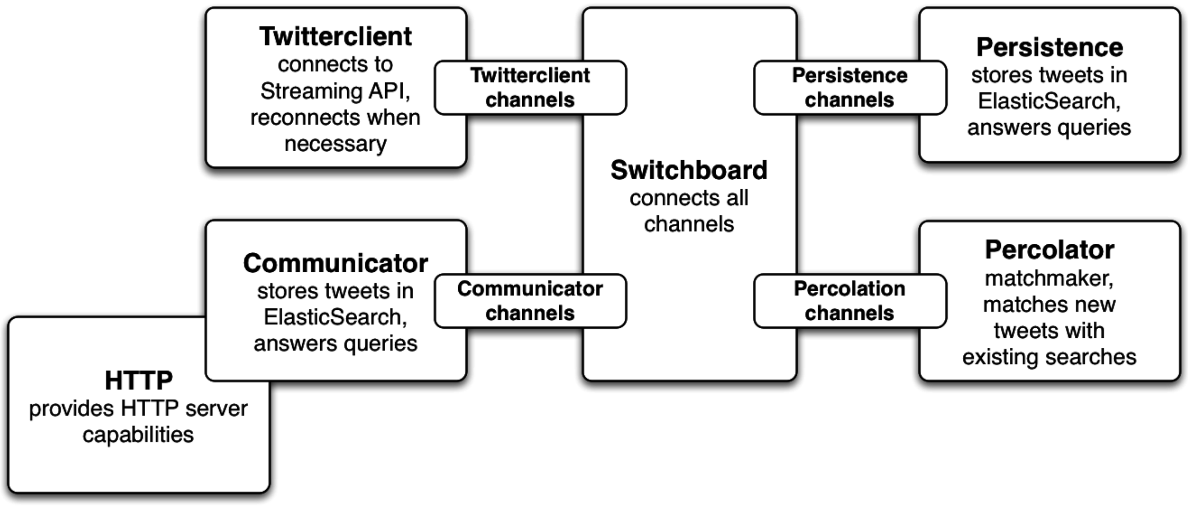
Cheers, Matthias

2. Old Server-side Architecture

2.1 Architectural Overview

In the following chapters, we will look at an earlier version of my **BirdWatch** application. The approach outlined here is not my recommended approach any longer, as there were a lot of functionalities that were common to each and all the components in the system. Instead, I have moved this common functionality into a separate library called **systems-toolbox**. However, by going through the initial approach, you will have a good reference for why I wanted to have the library. In particular, please look out for repetitiveness and try to think how you may get rid of it. Then later on in this book, we can compare notes as I will present what I did. Please also email me (the address is on my **GitHub** page) and discuss what you would do differently.

Now let's start with the basic architecture of the server side. Here's an overview:


Twitterclient
connects to Streaming API, reconnects when necessary
Twitterclient channels
Switchboard
connects all channels
Persistence channels
Persistence
stores tweets in ElasticSearch, answers queries
Communicator
stores tweets in ElasticSearch, answers queries
Communicator channels
Percolation channels
Percolator
matchmaker, matches new tweets with existing searches
HTTP
provides HTTP server capabilities

channels overview

You can see an animated version of this drawing in the **original blog post** that demonstrates how components in the system get wired up when the application initializes.

In the initial version that I wrote, where everything depended on everything, things were very different. Some people would call that "spaghetti code", but I think that is not doing justice to spaghetti. Unlike bad code, I don't mind touching spaghetti. I would rather liken bad code to hairballs, of the worst kind that is. Have you ever experienced the following: you are standing in the shower and the water doesn't drain. You notice something in the sink, so you squat down to pull it out only to start screaming, "Oh my god, it's a dead rat" a second later. I am referring to that kind of entangled hairball mess.

This is where dependency injection comes in. Can we agree that we don't like hairballs? Good. Usually, what we are trying to achieve is a so-called inversion of control, in which a component of the application knows that it will be injected something which implements a known interface at runtime. Then, no matter what the actual implementation is, it knows what methods it can call on that something because of the implemented interface.

Here, unlike in object-oriented dependency injection, things are a little different because we don't really have objects. The components play the role of objects, but as a further way of decoupling, I wanted them to only communicate via **core.async** channels. Channels are a great abstraction. Rich Hickey likens them to conveyor belts onto which you put something without having to know at all what happens on the other side. We will have a more detailed look at the channels in the next article. For now, as an abstraction, we can think about the channel components (the flat ones connecting the components with the switchboard) as **wiring harnesses**, like the one that connects the electronics of your car to your engine. The only way to interface with a modern engine (that doesn't have separate mechanical controls) is by connecting to this wiring harness and either send or receive information, depending on the channel / cable that you interface with.

2.2 Scaling out

I recently did a lot of transcript proofreading concerning enlightening talk for my **talk-transcripts project**. The most recent one was **Design, Composition and Performance**, the tenth transcript of a talk by **Rich Hickey**. That one in particular got me thinking that there are quite a few ideas I want to adopt, so it was time to make up my mind what exactly it is that I am trying to solve with this application.

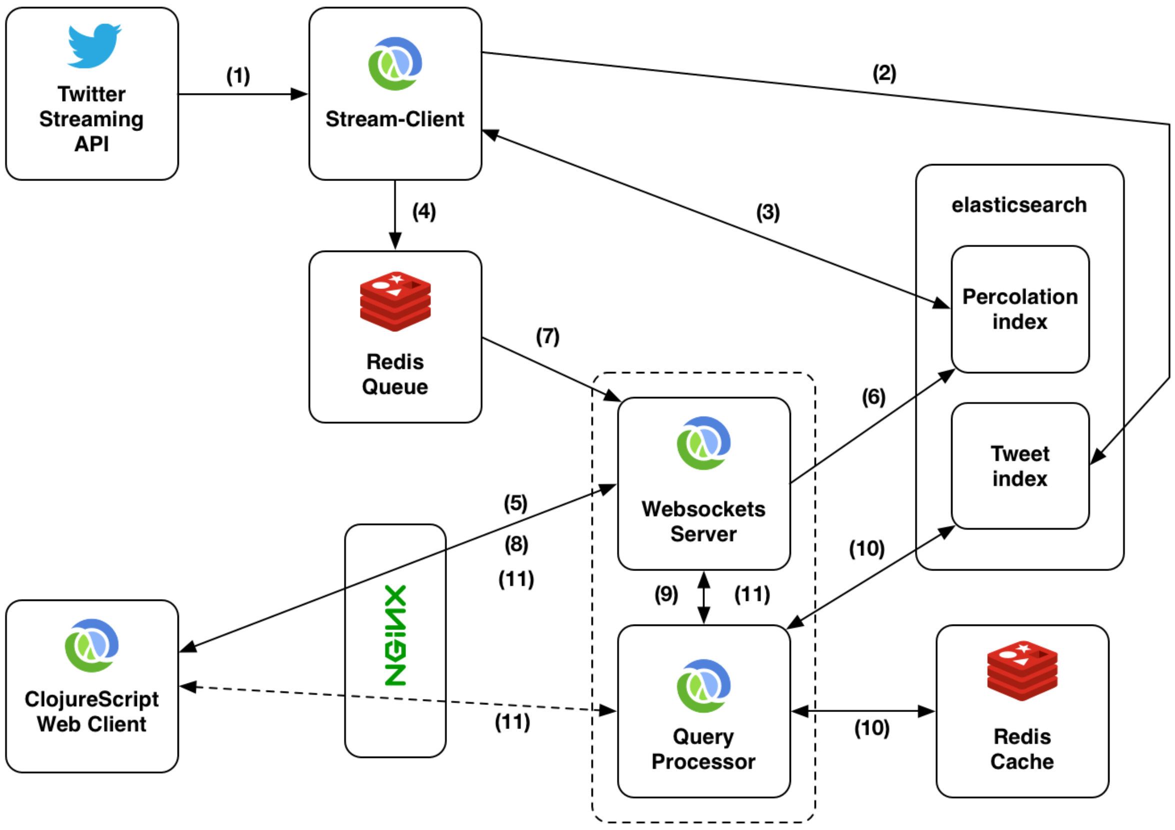
So here's the idea:

- We have a stream of information and we are interested in a subset of that information, which we can match on via **full-text search** and **ranges**. The searches are anything that **ElasticSearch** / **Lucene** can match on.
- Furthermore, we are interested in **live results** plus a certain period of time back into the **recent** past. For now, we are using tweets from the **Twitter Streaming API**, but the source could be anything, such as other social media data. Sensor data could also be really interesting. Live means new matches are added to the displayed results within about a second.
- The results are supposed to be shown in a browser, including on **mobile devices**. The number of items reasoned about should **not be limited by** the available **memory** of the browser[1].
- My next goal is to be able to reason about the **last one million tweets** for a certain topic. Also, it should be possible to serve **many concurrent ad-hoc queries**, like hundreds or more different ones.

What comes to mind immediately when regurgitating the requirements above is **Storm** and the **Lambda Architecture**. First I thought, great, such a search could be realized as a **bolt** in Storm. But then I realized, and please correct me if I'm wrong, that topologies are fixed once they are running. This limits the flexibility to add and tear down additional live searches. I am afraid that keeping a few stand-by bolts to assign to queries dynamically would not be flexible enough.

[1]Right now with all tweets loaded onto the client, the maximum for a desktop browser is somewhere in the range of a **few tens of thousands** of tweets before the application slows down noticeably.

So instead I suggest doing the **final aggregation** (the reduce phase) on the browser side in a **ClojureScript** application. On the server side, partial results are aggregated for shorter time periods. These partial results can be generated in a cluster of nodes whilst the client is fed with live data immediately. Let's have a look at a drawing before I walk you through the individual steps:



Redesigned Architecture

Initially, this application was designed to run in a single JVM but after some redesign, I split the server side into two different components. The architecture you saw in the drawing above turned out to be a great preparation for separating parts of the application into separate JVMs for scaling out the application. My idea was that there should be one **TwitterClient** application (as Twitter only allows one connection to the capped Streaming API) but multiple web client facing applications so that the number of connections the system could handle would not be limited by whatever a single machine could handle. Let's call that the **MainApp** application. The separation of components and limiting their interaction to passing messages on channels made it extremely simple to scale the application by fanning out the streaming data to multiple client-facing JVMs. All that was needed in addition was a component for interoperability. None of the existing components needed to be changed except for the switchboard where the data flow gets wired together.

Let's walk through the interactions of the entire system step by step:

1. Tweets are received from the Twitter Streaming API in chunks of (oftentimes incomplete) JSON. A **stateful transducer** is used to reassemble the JSON and parse chunks into Clojure

maps.

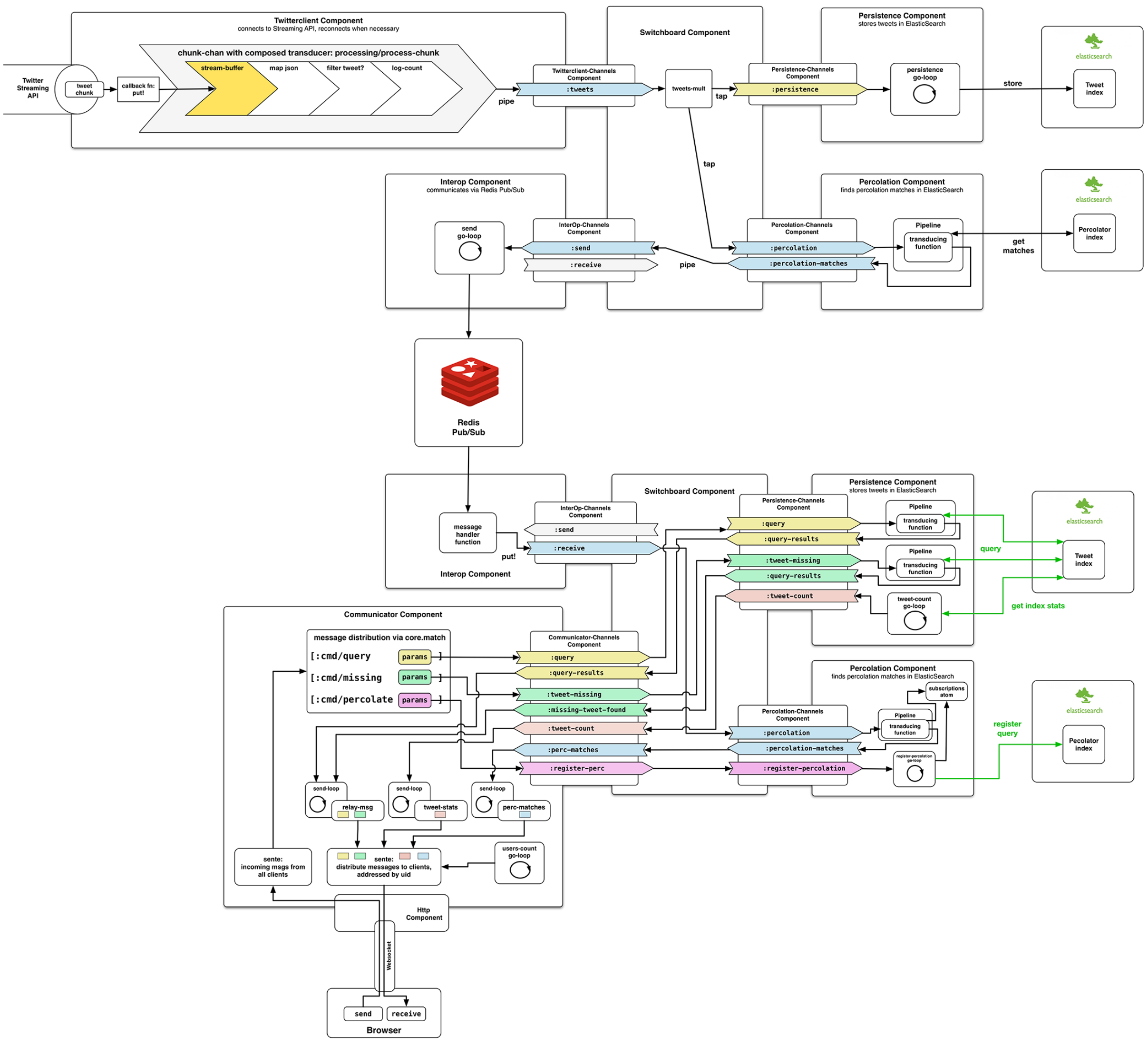
2. Tweets are stored in **ElasticSearch** in their respective index. If the received tweet contains a retweet, the retweet status will be used to update an existing item (e.g. the RT count).
3. The newly received tweet is presented to ElasticSearch's **percolation index** in order to find clients interested in this tweet. It is kind of a reverse matching where the new item is matched against **existing searches**. We will cover percolation in more detail when we look at the percolation component.
4. Together with information on matched queries the tweet is published using **Redis's Pub/Sub** feature. Potentially, the search ID of the matches could be used to publish to different topics[2]. This constitutes the border of the first Clojure application.
5. The second Clojure application, which serves the client-side ClojureScript application as well, receives a new search via a **WebSocket** connection.
6. It then **registers** the query in ElasticSearch's **percolation** index.
7. Next, the socket connection **subscribes** to the search ID's topic in Redis's **Pub/Sub** feature.
8. From now on matches to the client's search will be delivered immediately to the client-side ClojureScript application.
9. The idea is to **aggregate data on the server side** and only deliver the aggregated data structures back to the client side. For example, this could be a few hundred aggregates over increments of five minutes each. These increments can easily be made addressable (and cacheable): let's say it is 1:17pm. Then, we have a last and incomplete increment from 1:15pm that will be added on in the browser whereas all previous ones are complete and fixed. By treating the complete ones as **immutable**, we can cache them and forego unnecessary and expensive requests to ElasticSearch. Since these immutable previous chunks can be addressed individually, it may make sense to deliver them through REST endpoints instead of via the WebSocket connection (the dashed line)[3]. This is not implemented yet; instead, the client requests chunks of previous tweets and all reasoning about them happens in the browser.
10. We've already established that previous chunks can be cached. **Redis** seems like a great match utilizing the **EXPIRE feature**. So Redis would be queried for the presence of a certain chunk first. If it exists, it will be delivered right away. If not, ElasticSearch will be queried and the result will be delivered and stored in Redis for the next couple of hours or so to avoid unnecessary load on the ElasticSearch cluster. Currently, here we only query ElasticSearch without caching.
11. Finally, the aggregate will be delivered to the client. This could either be through the WebSocket connection or through **REST** (the dashed line). Currently, a (larger) chunk with 500 tweets each is delivered to the client instead of an aggregate.

So far, these changes have only been partially implemented. **Decoupling** the processes between a Twitter client and the client-serving part is done and allows restarting the latter **without**

[2]I'm undecided about this one. On one hand, it is strikingly simple to have a topic per search ID, which is a hash of the query itself. But on the other hand, this likely involves **book-keeping** of the subscriptions on the consuming side, where the subscription would have to be removed once the client disconnects. Maybe it is simpler to just serialize a set of IDs with the tweet and publish that on a single topic.

[3]Using REST makes communication somewhat more complex, but I still think it would make sense to move this aspect of the application into separate JVMs. The GC **characteristics** of aggregating large amounts of data in spikes are vastly different from the (near-) real-time aspects of the WebSocket delivery of new tweets. For the aggregation, it would not affect user experience much if there was a **stop-the-world** garbage collection pause of even a few seconds, but I don't want that to happen for the streaming data.

disconnecting from the Streaming API and also allows horizontal scaling where multiple client-serving applications can connect to the Pub/Sub:



Redesigned Architecture

The server-side aggregation has not been implemented yet, that part will follow soon.

2.3 Application Initialization

Let's have a look at how the initialization of the application looks like in code:

```
(ns birdwatch.main
  (:gen-class)
  (:require
   [birdwatch.twitter-client :as tc]
   [birdwatch.communicator :as comm]
   [birdwatch.persistence :as p]
   [birdwatch.percolator :as perc]
   [birdwatch.http :as http]
   [birdwatch.switchboard :as sw]
   [clojure.edn :as edn]
   [clojure.tools.logging :as log]
   [clj-pid.core :as pid]
   [com.stuartsierra.component :as component]))

(def conf (edn/read-string (slurp "twitterconf.edn")))

(defn get-system [conf]
  "Create system by wiring individual components so that component/start
  will bring up the individual components in the correct order."
  (component/system-map
   :communicator-channels (comm/new-communicator-channels)
   :communicator  (component/using (comm/new-communicator) {:channels :communicator-channels})
   :twitterclient-channels (tc/new-twitterclient-channels)
   :twitterclient (component/using (tc/new-twitterclient conf) {:channels :twitterclient-channels})
   :persistence-channels (p/new-persistence-channels)
   :persistence   (component/using (p/new-persistence conf) {:channels :persistence-channels})
   :percolation-channels (perc/new-percolation-channels)
   :percolator    (component/using (perc/new-percolator conf) {:channels :percolation-channels})
   :http          (component/using (http/new-http-server conf) {:communicator :communicator})
   :switchboard   (component/using (sw/new-switchboard) {:comm-chans :communicator-channels
                                                         :tc-chans :twitterclient-channels
                                                         :pers-chans :persistence-channels
                                                         :perc-chans :percolation-channels})))
(def system (get-system conf))

(defn -main [& args]
  (pid/save (:pidfile-name conf))
  (pid/delete-on-shutdown! (:pidfile-name conf))
  (log/info "Application started, PID" (pid/current))
  (alter-var-root #'system component/start))
```

I personally think this **reads really well**, even if you have never seen Clojure before in your life. Roughly the first half is concerned with imports and reading the configuration file. Next, we have the `get-system` function which declares what components depend on what other components. The system is finally started in the `-main` function (plus the process ID logged and saved to a file). This is all there is to know about the application entry point.

Now, when we start the application, all the dependencies will be started in an order that the component library determines so that all dependencies are met. Here's the output of that startup process:

```
mn:Clojure-Websockets mn$ lein run
16:46:30.925 [main] INFO  birdwatch.main - Application started, PID 6682
16:46:30.937 [main] INFO  birdwatch.twitter-client - Starting Twitterclient Channels Component
16:46:30.939 [main] INFO  birdwatch.twitter-client - Starting Twitterclient Component
16:46:30.940 [main] INFO  birdwatch.twitter-client - Starting Twitter client.
16:46:31.323 [main] INFO  birdwatch.persistence - Starting Persistence Channels Component
16:46:31.324 [main] INFO  birdwatch.persistence - Starting Persistence Component
16:46:31.415 [main] INFO  org.elasticsearch.plugins - [Chameleon] loaded [], sites []
16:46:32.339 [main] INFO  birdwatch.communicator - Starting Communicator Channels Component
16:46:32.340 [main] INFO  birdwatch.communicator - Starting Communicator Component
16:46:32.355 [main] INFO  birdwatch.http - Starting HTTP Component
16:46:32.375 [main] INFO  birdwatch.http - Http-kit server is running at http://localhost:8888/
16:46:32.376 [main] INFO  birdwatch.percolator - Starting Percolation Channels Component
16:46:32.377 [main] INFO  birdwatch.percolator - Starting Percolator Component
16:46:32.380 [main] INFO  birdwatch.switchboard - Starting Switchboard Component
```

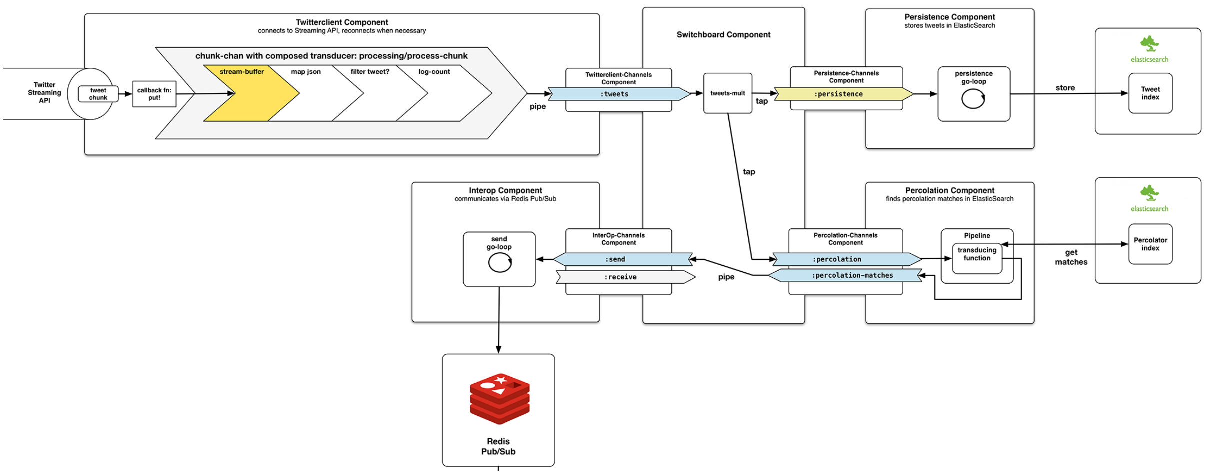
Note that in the code above, we are looking at a previous version of the application in which the entire application used to live inside a single JVM. The reason is that the corresponding animation in the **original blog post** also deals with this single-JVM architecture.

However, the mechanism remains the same in both applications after splitting the server side into multiple processes, so we only handle this once for now.

3. Server-side: TwitterClient

3.1 Architectural Overview

Here's an overview of the TwitterClient application. No worries if it's too small to read everything, we will look at different parts of it in detail later.



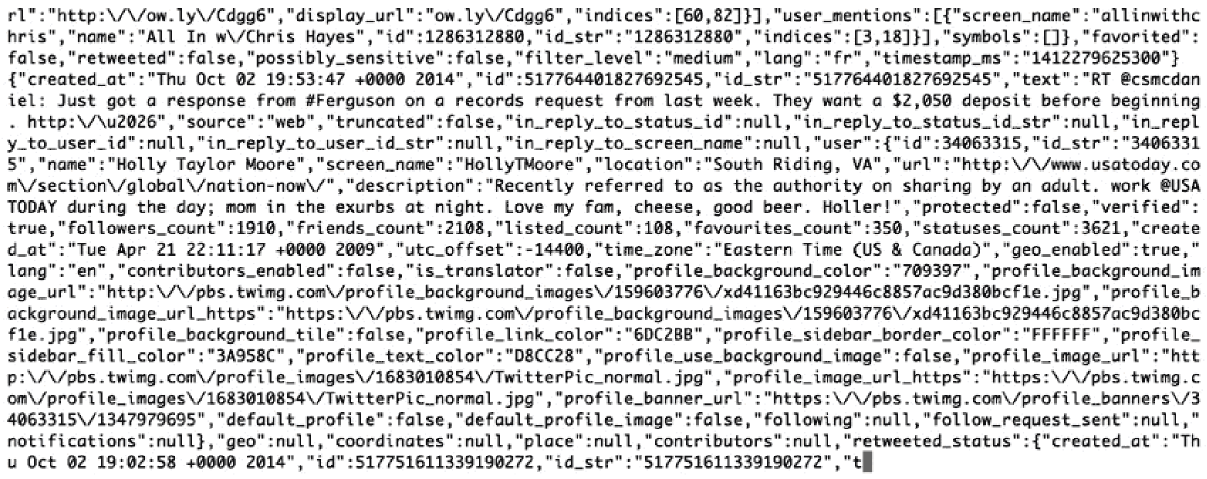
TwitterClient Application

The purpose of the **TwitterClient** application is to maintain a streaming connection with the **Twitter Streaming API**, restart this connection if necessary, persist received tweets and make tweets available on a **Redis Pub/Sub**. There can only be one instance of this application at any one time because Twitter does not allow you to start multiple clients at the same time.

Let's start in **hammock mode**, without code. What is the problem we are trying to solve? It all starts with the tweet stream from the Twitter API. Very briefly, the **Twitter Streaming API** allows us to subscribe to a (near) real time stream of tweets that contain one or more terms out of a set of terms. In the live instance under **http://birdwatch2.matthiasnehlsen.com** these terms at the moment happen to be "Ferguson", "ISIS", and "Ebola" - I am interested in all these topics. As long as that subscription does not hit a hard ceiling of 1% of all the tweets flowing through twitter's system, we can be sure that we will retrieve all of them. Otherwise the stream will be throttled to a maximum of 1% of what is tweeted at any moment in time. [1]

Here is how that stream looks like when each chunk is simply printed to the console:

[1]I don't know much about the exact mechanism at play, actual numbers or delivery guarantees. It anyhow doesn't matter much for the purpose of this application. The interesting views focus on the most retweeted tweets. Now every retweet contains the original tweet under "retweeted_status", with the current numbers such as retweet and favorite count for the moment in time it was retweeted. For popular ones, we thus receive the original tweet many, many times over. So even if we missed as much as half of all the tweets - which I consider unlikely - the popular tweets would only be updated less often. Worst case: retweet count is off by one or two. I can live with that. In reality, for the current selection of terms, reaching the limit also hardly ever happens. After all, 1% is still millions of tweets per day.



```
rl":"http:\/\/ow.ly\/Cdgg6","display_url":"ow.ly\/Cdgg6","indices":[60,82]}],"user_mentions":[{"screen_name":"allinwithc
hris","name":"All In w\/Chris Hayes","id":1286312880,"id_str":"1286312880","indices":[3,18]}],"symbols":[]},"favorited":
false,"retweeted":false,"possibly_sensitive":false,"filter_level":"medium","lang":"fr","timestamp_ms":"1412279625300"}
{"created_at":"Thu Oct 02 19:53:47 +0000 2014","id":517764401827692545,"id_str":"517764401827692545","text":"RT @csmcdan
iel: Just got a response from #Ferguson on a records request from last week. They want a $2,050 deposit before beginning
. http:\/\u2026","source":"web","truncated":false,"in_reply_to_status_id":null,"in_reply_to_status_id_str":null,"in_repl
y_to_user_id":null,"in_reply_to_user_id_str":null,"in_reply_to_screen_name":null,"user":{"id":34063315,"id_str":"3406331
5","name":"Holly Taylor Moore","screen_name":"HollyTMoore","location":"South Riding, VA","url":"http:\/\/www.usatoday.co
m\/section\/global\/nation-now\/","description":"Recently referred to as the authority on sharing by an adult. work @USA
TODAY during the day; mom in the exurbs at night. Love my fam, cheese, good beer. Holler!","protected":false,"verified":
true,"followers_count":1910,"friends_count":2108,"listed_count":108,"favourites_count":350,"statuses_count":3621,"create
d_at":"Tue Apr 21 22:11:17 +0000 2009","utc_offset":-14400,"time_zone":"Eastern Time (US & Canada)","geo_enabled":true,"
lang":"en","contributors_enabled":false,"is_translator":false,"profile_background_color":"709397","profile_background_im
age_url":"http:\/\/pbs.twimg.com\/profile_background_images\/159603776\/xd41163bc929446c8857ac9d380bcf1e.jpg","profile_b
ackground_image_url_https":"https:\/\/pbs.twimg.com\/profile_background_images\/159603776\/xd41163bc929446c8857ac9d380bc
f1e.jpg","profile_background_tile":false,"profile_link_color":"6DC2BB","profile_sidebar_border_color":"FFFFFF","profile_
sidebar_fill_color":"3A958C","profile_text_color":"D8CC28","profile_use_background_image":false,"profile_image_url":"htt
p:\/\/pbs.twimg.com\/profile_images\/1683010854\/TwitterPic_normal.jpg","profile_image_url_https":"https:\/\/pbs.twimg.c
om\/profile_images\/1683010854\/TwitterPic_normal.jpg","profile_banner_url":"https:\/\/pbs.twimg.com\/profile_banners\/3
4063315\/1347979695","default_profile":false,"default_profile_image":false,"following":null,"follow_request_sent":null,"
notifications":null},"geo":null,"coordinates":null,"place":null,"contributors":null,"retweeted_status":{"created_at":"Th
u Oct 02 19:02:58 +0000 2014","id":517751611339190272,"id_str":"517751611339190272","t
```

streaming API output

For reasons unbeknownst to me, tweets stopped respecting the chunk borders for the last half year. Instead, tweets occasionally span two or three chunks. This makes processing the tweets a little more complicated than we might wish for. One tweet per chunk is straightforward:

```
Receive chunk -> parse JSON into map -> put on conveyor belt (channel)
```

That looks like functional programming, right? No state to be kept anywhere, just functions producing results that are passed into other functions. But as desirable as that sounds, it does not align with reality. Instead, we need logical reasoning and state. What is the instruction we would give a sentient being? Imagine an intelligent agent standing between two conveyor belts. Imagine that agent being you. Here we go:

"On your left side, there's a conveyor belt that keeps delivering hundred dollar bills. Put all of them on the other conveyor belt. Some of them come out cut into multiple pieces. These fragments are in correct order. Scotch tape is over there."

I think we would all know what to do. There is a space where you park fragments of not-yet-complete bills / tweets. Then, with every new fragment, you inspect if the bill is complete and if so, put it back together and pass it on. Let's try that in code. First, we will need to introduce **transducers** though.

3.1.1 Transducers

> Transducers are a powerful and composable way to build algorithmic transformations that you can reuse in many contexts, and they're coming to Clojure core and core.async. Rich Hickey, August 2014

In a way, a transducer is the **essence** of a computation over data, without being bound to any kind of collection or data structure. Above, before we had to concern ourselves with the incomplete fragments, there was one step of the computation that we could **model as a transducer**: the part where we wanted to parse JSON into a map data structure.

Imagine we wanted to transform a vector of JSON strings into a vector of such parsed maps. We could simply do this:

```
(map json/read-json ["{\"foo\":1}" "{\"bar\":42}"])
```

However, the above is bound to the data structure, in this case a vector. That should not have to be the case, though. Rich Hickey provides a good example in his **transducers talk**, likening the above to having to tell the guys processing luggage at the airport the same instructions twice, once for trolleys and again for conveyor belts, where in reality that should not matter.

We could, for example, not only run the mapping function over every item in a vector but also reuse the same function on every item in a channel, stream or whatever.

With Clojure 1.7, we can now create such a transducing function by simply leaving out the data structure:

```
(def xform (map json/read-json))
```

Now, we can apply this transducing function to different kinds of data structures that are transducible processes. For example, we could transform all entries from a vector into another vector, like so:

```
(into [] xform ["{\"foo\":1}" "{\"bar\":42}"])
```

Or into a sequence, like this:

```
(sequence xform ["{\"foo\":1}" "{\"bar\":42}"])
```

It may not look terribly useful so far. But this can also be applied to a channel. Say, we want to create a channel that accepts JSON strings and transforms each message into a Clojure map. Simple:

```
(chan 1 xform)
```

The above creates a channel with a buffer size of one that applies the transducer to every element.

3.2 TwitterClient Component

But these simple, per-item transducers do not help in our case, where we know that some of the chunks are not complete but instead have to be glued together with the next one or two pieces. For that, we will need some kind of **state**. In the example above, that would be the space where we place fragments of a hundred dollar bill. But what if we want to see this aggregation process as a **black box**? Then, the aggregation cannot really have outside state. Also, as Rich Hickey mentioned in his StrangeLoop talk, there is no space in the machinery to keep state. What if one such transducer could have local state even if that is contained and not accessible from the outside? It turns out this is where stateful transducers can help.

Here's how this stateful transducer looks like in **code**:

```
(defn- streaming-buffer []
  (fn [step]
    (let [buff (atom "")]
      (fn
        ([r] (step r))
        ([r x]
         (let [json-lines (-> (str @buff x) (insert-newline) (str/split-lines))
               to-process (butlast json-lines)]
           (reset! buff (last json-lines))
           (if to-process (reduce step r to-process) r)))))))
```

Let's go through this line by line. We have a (private) function named **streaming-buffer** that does not take any arguments. It returns a function that accepts the step function. This step function is the function that will be applied to every step from then on. This function then first creates the local state as an atom[2] which we will use as a buffer to store incomplete tweet fragments. It is worth noting that we don't have to use **atoms** here if we want to squeeze out the last bit of performance, but I find it easier not to introduce yet another concept unless absoletely necessary[3]. Next, this function returns another function which accepts two parameters, r for result and x for the current data item (in this case the - potentially incomplete - chunk).

In the first line of the let binding, we use the `->` **(thread-first)** macro. This macro makes the code more legible by simply passing the result of each function call as the first argument of the next function. Here, specifically, we **1)** concatenate the buffer with the new chunk, **2)** add newlines where missing[4], and **3)** split the string into a sequence on the line breaks.

Now, we cannot immediately process all those items in the resulting sequence. We know that all are complete except for the last one as otherwise there would not have been a subsequent tweet. But the last one may not be complete. Accordingly, we derive

```
(butlast json-lines)
```

under the name **to-process**. Then, we reset the buffer to whatever is in that last string:

```
(reset! buff (last json-lines))
```

Finally, we have **reduce** call the **step** function for every item in **to-process**:

```
(if to-process (reduce step r to-process) r)
```

[2]After initial experimentation with a **local volatile reference**, I decided in favor of a good old atom. The **volatile!** local reference trades off potential race conditions with speed. But there's no performance issue when we process tweet chunks a few hundred times a second utmost, so why bother and introduce a new concept? Worth to keep in mind, though, when performance is an issue.

[3]**Atoms** are essential to Clojure's **state model**. Essentially, you have this managed reference that is thread-safe. Whenever we dereference such an atom, we get the state of the world this very second. Then, when you pass the dereferenced value to other parts of the application, it still represents the immutable state of the world at that point in time. It cannot change. Next time I dereference that atom, I will get the new state of the world. Updates to atoms can only happen in transactions, meaning that no two can run at the same time. Thus, we won't have to chase crazy concurrency issues.

[4]For whatever reason, the changed behavior of the streaming API also entails that not all tweets are followed by line breaks, only most of them. A tiny helper function inserts those missing linebreaks where they are missing between two tweets: `(str/replace s #"\}\{" "}\r\n{")`.

That way, only complete JSON strings are pushed down to the next operation, whereas intermediate JSON string fragments are kept locally and not passed on until certainly complete. That's all that was needed to make the tweets whole again. Next, we compose this with the JSON parsing transducer we have already met above so that this **streaming-buffer** transducer runs first and passes its result to the **JSON parser**.

Let's create a vector of JSON fragments and try it out. We have already established that transducers can be used on different data structures, it therefore should work equally well on a vector. Here's the vector for the test:

```
["{\"foo\"" ":1}\n{\"bar\":" "42}" "{\"baz\":42}" "{\"bla\":42}"]
```

Now we can check on the REPL if this will produce three complete JSON strings. It is expected here that the last one is lost because we would only check its completeness once there is a following tweet[5]. Once the collection to process is empty, the **arity-1** (single argument) function is called one last time, which really only returns the aggregate at that point:

```
=> (in-ns 'birdwatch.twitterclient.processing)
#<Namespace birdwatch.twitterclient.processing>

=> (def chunks ["{\"foo\"" ":1}\n{\"bar\":" "42}" "{\"baz\":42}" "{\"bla\":42}"])
#'birdwatch.twitterclient.processing/chunks

=> (into [] (streaming-buffer) chunks)
["{\"foo\":1}" "{\"bar\":42}" "{\"baz\":42}"]
```

What somewhat confused me at first is what the step function actually was. Let's find out by printing it when the arity-1 function is called. We can modify the fourth line of **stream-buffer** like this:

```
([r] (println step) (step r))
```

Now when we run the same as above on the REPL, we can see what the step function actually is:

```
=> (into [] (streaming-buffer) chunks)
#<core$conj_BANG_ clojure.core$conj_BANG_@5fd837a>
["{\"foo\":1}" "{\"bar\":42}" "{\"baz\":42}"]
```

Interestingly, the step function is **conj!** which according to the source adds **x** to a **transient collection**[6].

The step function is different when we use the transducer on a channel, but more about that when we use it in that scenario.

There's more to do before we can **compose all transducers** and attach them to the appropriate channel. Specifically, we can receive valid JSON from Twitter, which is not a tweet. This happens,

[5]One could probably check if the buffer contains a valid and complete JSON string when the arity-1 function is called, and if so, pass it on. Considering though that in this application we are interested in a stream that does not have an end, I omitted this step.

[6]I assume the **transient** collection is used for performance reasons.

for example, when we get a notification that we lag behind in consuming the stream. In that case we only want to pass on the parsed map if it is likely that it was a tweet and otherwise log it as an error. There is one **key** that all tweets have in common, which does not seem to appear in any status messages from Twitter: **:text**. We can thus use that key as the **predicate** for recognizing a tweet. Here's the **code**:

```
(defn- tweet? [data]
  "Checks if data is a tweet. If so, pass on, otherwise log error."
  (let [text (:text data)]
    (when-not text (log/error "error-msg" data))
    text))
```

Next, we also want to log the count of tweets received since the application started. Let's do this only for full thousands. We will need some kind of counter to keep track of the count. Let's create another **stateful transducer**:

```
(defn- log-count [last-received]
  "Stateful transducer, counts processed items and updating last-received atom. Logs progress every 1000 i\
tems."
  (fn [step]
    (let [cnt (atom 0)]
      (fn
        ([r] (step r))
        ([r x]
         (swap! cnt inc)
         (when (zero? (mod @cnt 1000)) (log/info "processed" @cnt "since startup"))
         (reset! last-received (t/now))
         (step r x))))))
```

This transducer is comparable to the one we saw earlier, except that the local atom now holds the count. Initially, the counter is incremented and then, when the counter is divisible by 1000, the count is logged. In addition, this function also resets the **last-received** timestamp. Of course, this could be factored out into a separate function, but I think this will do.

Now, we can compose all these steps:

```
(defn process-chunk [last-received]
  "Creates composite transducer for processing tweet chunks. Last-received atom passed in for updates."
  (comp
   (streaming-buffer)
   (map json/read-json)
   (filter tweet?)
   (log-count last-received)))
```

The above creates a composed function that takes the timestamp atom provided by the TwitterClient component as an argument. The entires namespace can be found **here**. We can now use this **transducing function** and apply it to different data structures. Here, we use it to create a channel that takes tweet chunk fragments and delivers parsed tweets on the other side of the conveyor belt.

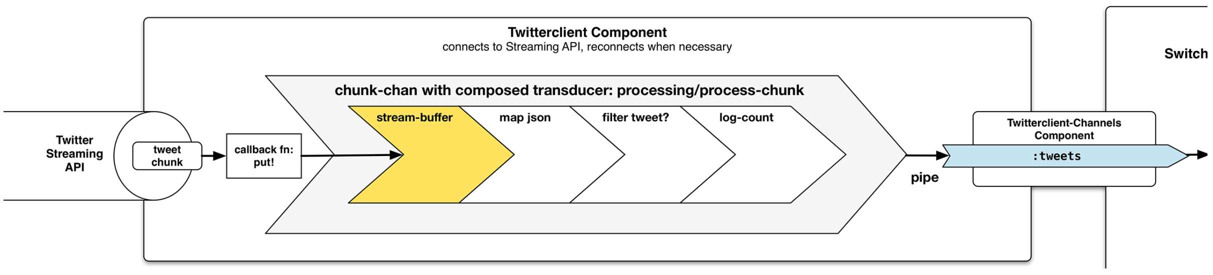
Let's try the composed transducer on a vector to see what's happening. For that, we create a vector with two JSON strings that contain the **:text** property and two that don't.

```
["{\"text\"" ":\"foo\"}\n{\"text\":" "\"bar\"}" "{\"baz\":42}" "{\"bla\":42}"])
```

Then we should see that the invalid one is logged and the other two are returned (the final one at that point still being in the buffer):

```
=> (in-ns 'birdwatch.twitterclient.processing)
=> (def chunks ["{\"text\"" ":\"foo\"}\n{\"text\":" "\"bar\"}" "{\"baz\":42}" "{\"bla\":42}"])
=> (into [] (process-chunk (atom (t/epoch))) chunks)
ERROR birdwatch.twitterclient.processing - error-msg {:baz 42}
[{:text "foo"} {:text "bar"}]
```

Great, we have a composed transducer that works on vectors as expected. According to Rich Hickey this should work equally well on channels. But let's not take his word for it and instead try it out. First, here's my attempt to visualize the usage of a transducer in a channel. To make things easier, no errors occur.


Twitterclient Component
connects to Streaming API, reconnects when necessary
chunk-chan with composed transducer: processing/process-chunk
stream-buffer
map json
filter tweet?
log-count
Twitter Streaming API
tweet chunk
callback fn: put!
pipe
Switch
Twitterclient-Channels Component
:tweets

TwitterClient Component with channels

You can also see the illustration above as an animation in the original blog post.

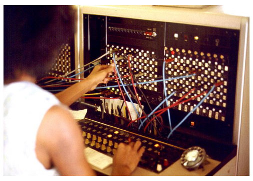
Now for a simple example in the REPL:

```
=> (in-ns 'birdwatch.twitterclient.processing)
#<Namespace birdwatch.twitterclient.processing>

=> (def chunks ["{\"text\"" ":\"foo\"}\r\n{\"text\":" "\"bar\"}" "{\"baz\":42}" "{\"bla\":42}"])
#'birdwatch.twitterclient.processing/chunks

=> (require '[clojure.core.async :as async :refer [chan go-loop <! put!]])
=> (def c (chan 1 (process-chunk (atom (t/now)))))
#'birdwatch.twitterclient.processing/c

=> (go-loop [] (println (<! c)) (recur))
#<ManyToManyChannel clojure.core.async.impl.channels.ManyToManyChannel@2f924b3f>

=> (put! c (chunks 0))
=> (put! c (chunks 1))
{:text foo}

=> (put! c (chunks 2))
=> (put! c (chunks 3))
{:text bar}

=> (put! c (chunks 4))
ERROR birdwatch.twitterclient.processing - error-msg {:baz 42}
```

Excellent, same output. In case you're not familiar with **core.async channels** yet: above we created a channel with the same transducer attached as in the previous example, then we created a `go-loop` to consume the channel and finally, we `put!` the individual chunks on the channel. No worries if this seems a little much right now. We'll cover this topic in much more detail in later chapters.

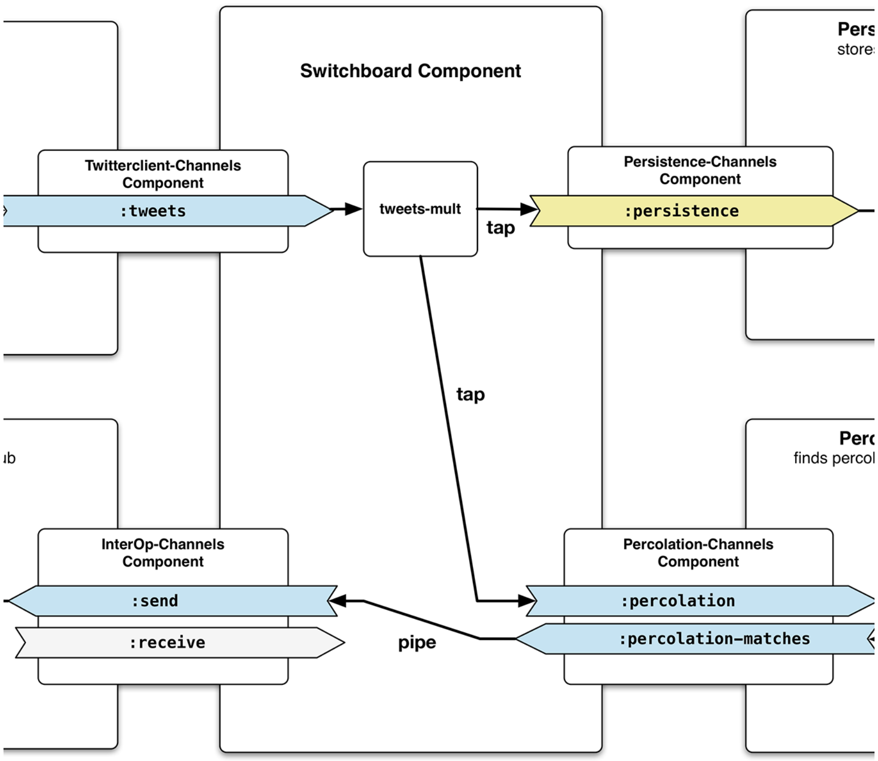
3.3 TwitterClient - SwitchBoard Component

This component takes care of distributing data in the TwitterClient application by connecting **core.async** channels together, comparable to an operator in the early days of telephony[7]:



telephony switchboard

In the conceptual drawing we can see that in this component the channels from different components are wired for the requirements of the application. Here, it would be very simple to attach additional components and send messages to those as well, without other parts of the application having to know anything about it.



Switchboard Component

[7]Photo by Joseph A. Carr, 1975, from **Wikimedia**

Let's have a look at the **code**:

```
(ns birdwatch-tc.switchboard
  (:gen-class)
  (:require
   [clojure.tools.logging :as log]
   [com.stuartsierra.component :as component]
   [clojure.core.async :as async :refer [chan mult tap pipe]]))

;;;; This component is the central switchboard for information flow in this application.
;;;; The individual channel components come together like wiring harnesses in a car.

(defrecord Switchboard [tc-chans pers-chans perc-chans iop-chans]
  component/Lifecycle
  (start [component] (log/info "Starting Switchboard Component")
         (let [tweets-mult (mult (:tweets tc-chans))]
           (tap tweets-mult (:percolation perc-chans))   ; Tweets are distributed to multiple channels
           (tap tweets-mult (:persistence pers-chans))   ; through tapping the mult created above
           ;; Connect channels 1 on 1. Here, it would be easy to add message logging.
           (pipe (:percolation-matches perc-chans) (:send iop-chans))
           (assoc component :tweets-mult tweets-mult)))

  (stop [component] (log/info "Stop Switchboard Component")
        (assoc component :tweets-mult nil)))

(defn new-switchboard [] (map->Switchboard {}))
```

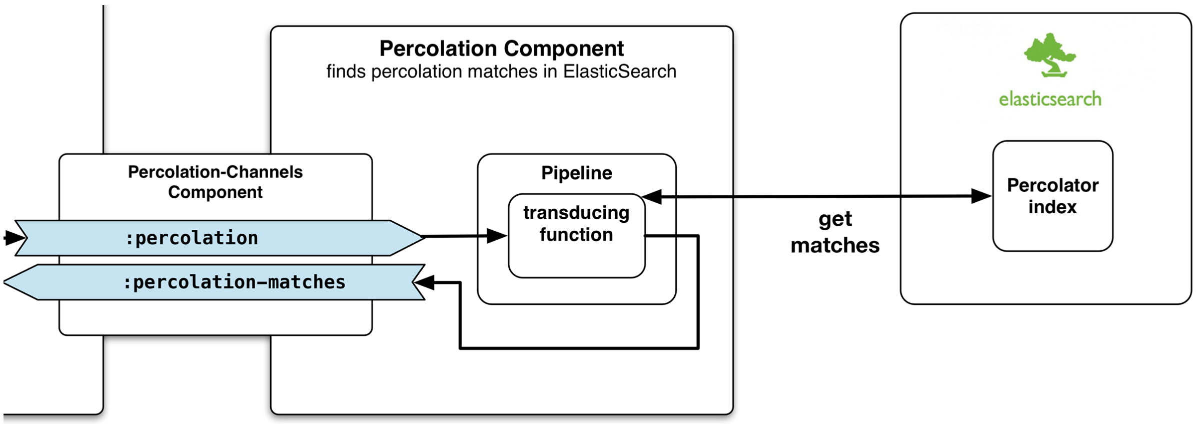
Here, the only thing that really happens is that we create a `mult` from the `:tweets` channel, which is a multiplier that allows us to connect multiple channels that each consume all elements on the `mult`. Then, we `tap` into the `mult` and connect both the `:percolation` and the `:persistence` channels.

3.4 TwitterClient - Percolation Component

The Percolation Component is responsible for matching new tweets with existing queries. Remember, in this application, we update the search results shown in the client in (near-)real time when new matches are available. In order to do that, we need some kind of matching mechanism between searches and new items.

This is where ElasticSearch's **Percolator feature** helps. Percolation queries are kind of reverse searches that allow the registration of an observing real-time search in the percolation index. Each new tweet is then presented to the percolation index in ElasticSearch to determine which of the registered searches match on the new item.

The registration of queries in the percolation index and the delivery happens in the percolation component of the **client-facing application** and will be covered in more detail there. Here, you just need to know that upon registering a search, a hash of the query is used as the ID so that any possible query is only ever registered once.



Percolator Component with Channels

In this component, new tweets are matched against existing searches, which returns a sequence of matching query IDs. New tweets are received on the `:percolation` channel and results (tweet with set of matches) are put on the `:percolation-matches` channel from the Percolation-Channels component. Here's the **component itself**:

```clojure
(ns birdwatch-tc.percolator.component
  (:gen-class)
  (:require
   [birdwatch-tc.percolator.elastic :as es]
   [clojure.tools.logging :as log]
   [pandect.core :refer [sha1]]
   [clojure.pprint :as pp]
   [clojurewerkz.elastisch.rest :as esr]
   [com.stuartsierra.component :as component]
   [clojure.core.async :as async :refer [chan tap pipeline-blocking]]))

(defrecord Percolator [conf channels]
  component/Lifecycle
  (start [component] (log/info "Starting Percolator Component")
         (let [conn (esr/connect (:es-address conf))
               perc-matches-chan (:percolation-matches channels)
               perc-chan (:percolation channels)]
           (pipeline-blocking 2 perc-matches-chan (es/percolation-xf conn) perc-chan)
           (assoc component :conn conn)))
  (stop [component] (log/info "Stopping Percolator Component") ;; TODO: proper teardown of resources
        (assoc component :conn nil)))

(defn new-percolator [conf] (map->Percolator {:conf conf}))

(defrecord Percolation-Channels []
  component/Lifecycle
  (start [component] (log/info "Starting Percolation Channels Component")
         (assoc component :percolation (chan) :percolation-matches (chan)))
  (stop [component] (log/info "Stop Percolation Channels Component")
        (assoc component :percolation nil :percolation-matches nil)))

(defn new-percolation-channels [] (map->Percolation-Channels {}))
```

The component follows the pattern of creating `defrecords` for the component itself plus an associated channels component in the same way that we've seen already. You may not have seen **pipeline-blocking** yet, so let me explain. A **pipeline** is a **core.async** construct that we can use when we want to take something off a channel, process it and put the result onto another channel, all potentially in parallel. In this case, we use two parallel blocking pipelines since querying ElasticSearch here is a blocking operation and we want to be able to run two in parallel at any moment.

All we need to supply to the pipeline is a **transducing function**, which we look at next:

```clojure
(ns birdwatch-tc.percolator.elastic
  (:gen-class)
  (:require
   [clojure.tools.logging :as log]
   [pandect.core :refer [sha1]]
   [clojure.pprint :as pp]
   [clojurewerkz.elastisch.rest.percolation :as perc]
   [clojurewerkz.elastisch.rest.response :as esrsp]))

(defn percolation-xf
  "create transducer for performing percolation"
  [conn]
  (map (fn [t]
         (let [response (perc/percolate conn "percolator" "tweet" :doc t)
               matches (set (map :_id (esrsp/matches-from response)))] ;; set with SHAs
           [t matches]))))
```

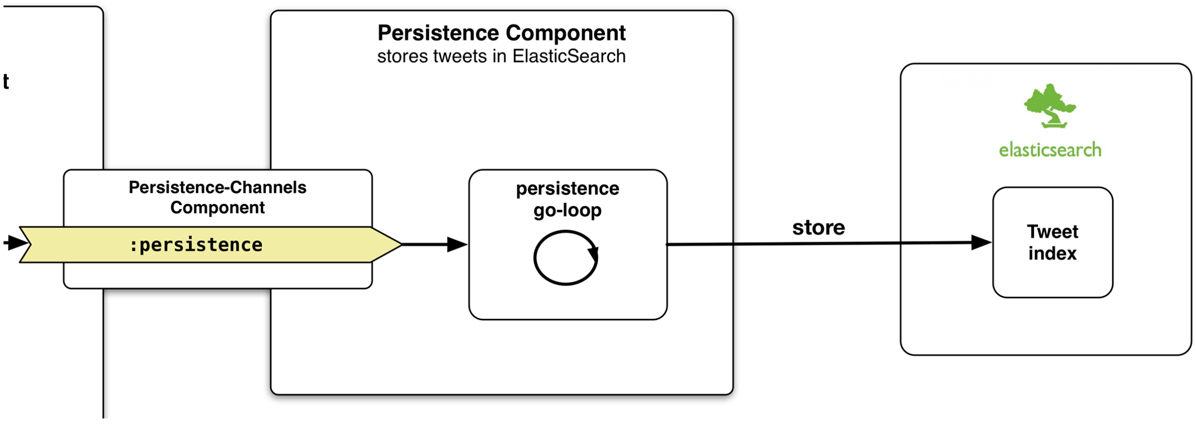
So here's what this function does. For every element (which we know is a tweet) that the pipeline construct processes by using the transducing function, we pass the item to the percolator, which first gives us a response. We then use `esrsp/matches-from` to retrieve the actual matches, use `map` to only get the `:_id` from each match and create a `set` from these matches.

Finally, we create a `vector` that contains the set and the actual tweet: `[t matches]`. This result vector is what we finally put on the `:percolation-matches` channel.

Note that this component knows nothing about any other part of the program. The transducer does not even know the target channel; it is only concerned with the actual processing step.

3.5 TwitterClient - Persistence Component

This component takes care of persisting tweets to an index in ElasticSearch. Once again we have a component with the typical lifecycle functions and an associated `Persistence-Channels` component. Here, this component only has a single channel, `:persistence`.



Persistence Component with Channels

Because there is only a channel to take from but no other channel to put a result onto, we will not use a `pipeline` but instead run a good old `go-loop`. Inside the **component**, there aren't any surprises:

```
(ns birdwatch-tc.persistence.component
  (:gen-class)
  (:require
   [birdwatch-tc.persistence.tools :as pt]
   [birdwatch-tc.persistence.elastic :as es]
   [clojure.tools.logging :as log]
   [clojure.pprint :as pp]
   [clojurewerkz.elastisch.rest :as esr]
   [com.stuartsierra.component :as component]
   [clojure.core.async :as async :refer [<! chan go-loop tap]]))

(defrecord Persistence [conf channels]
  component/Lifecycle
  (start [component]
         (log/info "Starting Persistence Component")
         (let [conn (esr/connect (:es-address conf))]
           (es/run-persistence-loop (:persistence channels) conf conn)
           (assoc component :conn conn)))
  (stop [component] ;; TODO: proper teardown of resources
        (log/info "Stopping Persistence Component")
        (assoc component :conn nil)))

(defn new-persistence [conf] (map->Persistence {:conf conf}))

(defrecord Persistence-Channels []
  component/Lifecycle
  (start [component] (log/info "Starting Persistence Channels Component")
         (assoc component :persistence (chan)))
  (stop [component] (log/info "Stop Persistence Channels Component")
        (assoc component :persistence nil)))

(defn new-persistence-channels [] (map->Persistence-Channels {}))
```

All we see above is yet another component that really only has the single channel `:persistence` inside the `Persistence-Channels` component and that **starts** said `go-loop` and passes the channel plus some configuration and the connection object.

```
(ns birdwatch-tc.persistence.elastic
  (:gen-class)
  (:require
   [clojure.tools.logging :as log]
   [clojure.pprint :as pp]
   [clojurewerkz.elastisch.rest :as esr]
   [clojurewerkz.elastisch.rest.document :as esd]
   [clojure.core.async :as async :refer [<! chan put! timeout go-loop]]))

(defn run-persistence-loop
  "run loop for persisting tweets"
  [persistence-chan conf conn]
  (go-loop [] (let [t (<! persistence-chan)]
                (try
                  (esd/put conn (:es-index conf) "tweet" (:id_str t) t)
                  (catch Exception ex (log/error ex "esd/put error"))))
           (recur)))
```

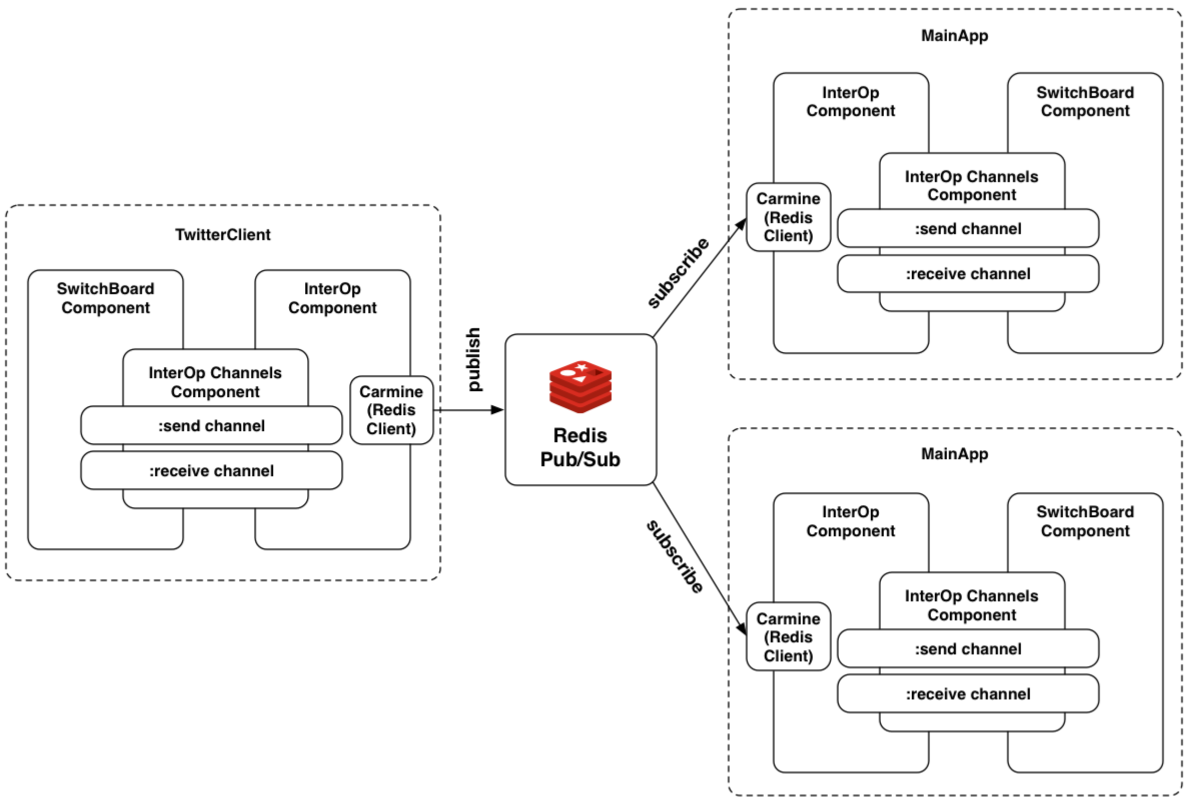
The `go-loop` above is pretty straightforward. Whatever we encounter on this channel, we try to persist in ElasticSearch, inside the index as specified in `(:es-index conf)`, for type `tweet`, with `(:id_str t)`as the **document id** and finally with the tweet `t` itself.

This makes me think that I've been wanting to implement **schema** for a while now. I don't mind using a plain old map to represent a tweet, but coming from strongly typed languages, I would at least like something to blow up when the item does not conform to an expected schema as opposed to storing something completely different while still calling its type in ElasticSearch `tweet`. Certainly an improvement to make soon.

Have you noticed a pattern? Once again, nothing in this component (and the associated namespace) knows anything about any other part of the application. The only thing I'd be okay with sharing in this context would be a schema for the tweet as mentioned above. The schema should then be maintained in one place for the entire system.

3.6 TwitterClient - InterOp Component

In the "Scaling Out" section of the architectural overview, I drew a picture of how I wanted to break apart the initial monolithic application and instead run different parts of the application in separate processes / separate JVMs. The idea was to have a single client for the connection to the **Twitter Streaming API** and the persistence of the received tweets in **ElasticSearch**, plus multiple machines to serve WebSocket connections to the client. For the communication between the processes, I picked **Redis Pub/Sub** because its model of communication suits the requirements really well.



Redesigned Architecture - InterOp

3.6.1 Redis Pub/Sub with Carmine

I chose **Pub/Sub** over a queue because I wanted to **fan-out** messages to multiple clients. Any connected processes are only supposed to be fed with data during their uptime, with no need to store anything for when they aren't connected. For interfacing with **Redis** from Clojure, I then chose **Peter Taoussanis**'s **carmine** client and it turned out to be a great choice.

Let's look at some code. First of all, I am using a **component** that provides a **send channel** and a **receive channel**. It can be reused on either side of the Pub/Sub connection (or for bidirectional communication, of course). Here's the **code**.

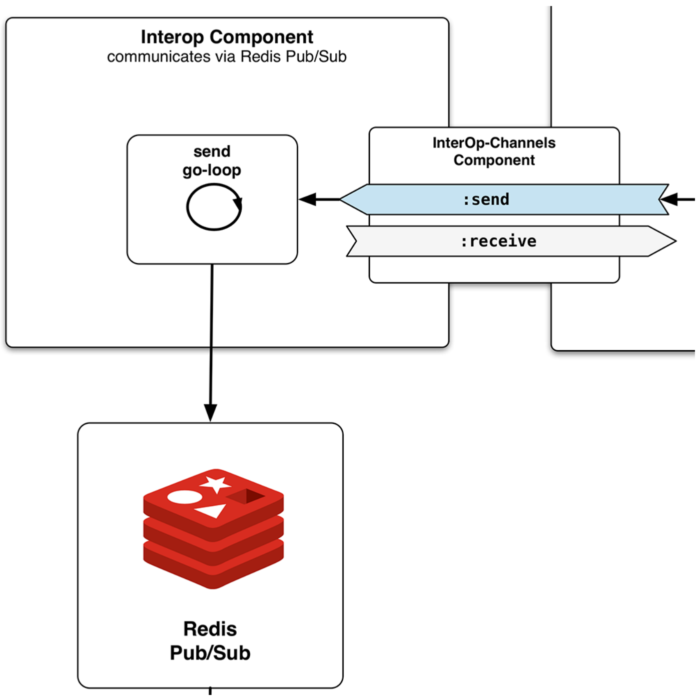
```clojure
(ns birdwatch-tc.interop.component
  (:gen-class)
  (:require
   [birdwatch-tc.interop.redis :as red]
   [clojure.tools.logging :as log]
   [clojure.pprint :as pp]
   [com.stuartsierra.component :as component]
   [clojure.core.async :as async :refer [chan]]))

;;; The interop component allows sending and receiving messages via Redis Pub/Sub.
;;; It has both a :send and a :receive channel and can be used on both sides of the Pub/Sub.
(defrecord Interop [conf channels]
  component/Lifecycle
  (start [component] (log/info "Starting Interop Component")
         (let [conn {:pool {} :spec {:host (:redis-host conf) :port (:redis-port conf)}}]
           (red/run-send-loop (:send channels) conn "matches")
         (assoc component :conn conn)))
  (stop  [component] (log/info "Stopping Interop Component") ;; TODO: proper teardown of resources
         (assoc component :conn nil)))

(defn new-interop [conf] (map->Interop {:conf conf}))

(defrecord Interop-Channels []
  component/Lifecycle
  (start [component] (log/info "Starting Interop Channels Component")
         (assoc component :send (chan) :receive (chan)))
  (stop  [component] (log/info "Stop Interop Channels Component")
         (assoc component :send nil :receive nil)))

(defn new-interop-channels [] (map->Interop-Channels {}))
```

The `Interop-Channels` component can now be wired into the `Interop` component where we create a configuration map and start a send loop with this configuration for the “**matches**” topic. Here’s that **run-send-loop** function:

```
(ns birdwatch-tc.interop.redis
  (:gen-class)
  (:require
   [clojure.tools.logging :as log]
   [clojure.pprint :as pp]
   [clojure.core.match :as match :refer (match)]
   [taoensso.carmine :as car :refer (wcar)]
   [clojure.core.async :as async :refer [<! put! go-loop]]))

(defn run-send-loop
  "loop for sending items by publishing them on a Redis pub topic"
  [send-chan conn topic]
  (go-loop [] (let [msg (<! send-chan)]
                (car/wcar conn (car/publish topic msg))
                (recur))))

(defn- msg-handler-fn
  "create handler function for messages from Redis Pub/Sub"
  [receive-chan]
  (fn [[msg-type topic payload]]
    (when (= msg-type "message")
      (put! receive-chan payload))))

(defn subscribe-topic
  "subscribe to topic, put items on specified channel"
  [receive-chan conn topic]
  (car/with-new-pubsub-listener
    (:spec conn)
    {"matches" (msg-handler-fn receive-chan)}
    (car/subscribe topic)))

(defn unsubscribe
  "unsubscribe listener from all topics"
  [listener]
  (car/with-open-listener listener (car/unsubscribe)))

(defn close
  "close listener"
  [listener]
  (car/close-listener listener))
```

Here, we are only using `run-send-loop` to start sending all messages that come in on a channel. Specifically, this`go-loop` consumes all messages that come in on the `send-chan` channel and publishes them on the `topic` in Redis for the specified configuration connection `conn`. None of the other functions is used here, but this component is the same on both sides of the pub/sub. We will look at the counterpart when looking at the **MainApp** application.

Here's a drawing of this component together with the channels:



InterOp Component with Channels

3.6.2 Performance of Redis

Redis does a lot with very little CPU utilization. In a non-scientific test, I fired up 50 JVMs (on four machines) subscribing to the topic on which the TwitterClient publishes tweets with matched percolation queries. Then I changed the tracked term from the Twitter Streaming API to "**love**", which reliably maxes out the rate of tweets permitted. Typically, with this term I see around **60 to 70** tweets per second. With 50 connected processes, **3000 to 3500** tweets were delivered per second overall, yet the CPU utilization of Redis idled somewhere between **1.7%** and **2.3%**.

4. The systems-toolbox library

4.1 Introduction

Maybe you have read the previous chapters carefully, or maybe you just skimmed through. Either is fine as long as you noticed the **repetitiveness** in the code. If not, maybe it helps to squint when looking at the code. Don't get me wrong, I'm not saying the code is entirely bad or anything, and I believe one could write an application this way and get along okay. The **BirdWatch** application, for example, was running like this for many months without a glitch, much to my satisfaction, and processing tens of millions of tweets. I just didn't like it too much.

Writing about the architecture made the abundance of boilerplate painfully evident as I noticed that I was describing the same things over and again. Writing repetitive code is one thing, but writing ABOUT repetitive code is much worse. It's a good exercise, however. Try writing about some project and its architecture. If something annoys you, you may well be onto something.

So when going through this exercise, I made up my mind that I didn't want to repeat myself as much when building somewhat complex systems. Not in any of my toy projects and for sure not in commercial projects, either. But how could one extract the commonality between different parts of the system?

At the same time when I started looking at refactoring my applications, I had started looking at the world through the lens of **Systems Thinking**. There is something very compelling about seeing the world as a mesh of systems that potentially interact in one way or another. Coming from that perspective, I was wondering if I could apply that kind of thinking to software architecture where an application forms a system that consists of subsystems.

Such a subsystem could, for example, be the persistence layer that handles saving data to some storage. Or it could be the HTTP server. Or a UI component. Whatever, any entity that has definable borders and that serves an explicit purpose can be such a subsystem.

So what is the commonality of such subsystems or components? By the way, I will use the terms *subsystem* and *component* interchangeably. Let me start with a brief rant about **Object-Oriented Programming**. There's one thing I never got, and that's the fanfare about encapsulation. We typically argue that encapsulation is essential for protecting the mutable state of an object from accidental mutilation. Fair enough, if you buy into the idea of uncontained mutability. But coming from functional programming and wearing Clojure-colored glasses, I don't think application state should ever live inside any entity that allows direct and uncontrolled mutation from anywhere. Instead, immutable data structures are ALWAYS the better choice when trying to relay information in any way. But as long as those data structures are immutable, why should you and I not see every detail there is to see? If there's no harm in doing so, why not, right? Plus, additional information never hurt anyone when trying to reason about a system.

So with the above in mind, I'd like each subsystem to be fully observable. To be able to do useful stuff, such a subsystem would need to be able to retain some state, which I want to be able to inspect at any point during runtime. Such subsystem would also need the means to communicate with the rest of the world, and, in particular, other parts of the overall system via message passing, but, at the edges of the system as a whole, also with the outside world.

When it comes to relaying information, I found **Communicating Sequential Processes (CSP)** to be a very useful abstraction. However, directly modeling an application by utilizing **core.async**, Clojure's incarnation of CSP, always seemed a little tedious to me. Say you had a few different gadgets in a home automation system, such as the central heating, flow meters, thermometers, and remotely controlled thermostats, how do you map their interaction mentally? Do you think along the lines of, this connects to that in some way to achieve a meaningful purpose, or do you mentally lay out the actual plumbing or the network devices, wires or airwaves? I didn't think so, other than that you may want to know about the general properties of the connections involved.

I felt the same way about **core.async**. It's a very useful library, but the previous architecture of BirdWatch forced me to look at the actual plumbing where I only wanted to think about different subsystems and their actions and connections.

Now let's look at **subsystems** one more time. Any given component interacts with the outside world in two different ways. First, there's direct communication, where there's the expectation that the recipient of a message acts the way it wants to a given message. You know, like when you're sending your tax declaration to the appropriate authorities, such as the IRS. You expect a response. You better don't have specific expectations about how long it'll take, but at some point, you will get a response, and you will then (hopefully) know how to handle the message. In essence, that's asynchronous message passing. With this kind of message passing, there's a general expectation that the message enacts something, whatever that something is. Or, if no other subsystem is wired to react to such a message, we can expect nothing to happen, other than that message leaving some observable trace.

And then there's another kind of information passing altogether. Here, it's all about observability. When the thermometer on my balcony measures the temperature and displays it, it has no notion of wanting to enact anything in the real world. The only reason for the display of the temperature data lies in the observability of the reading. I as the observer then usually only care about the latest reading. The remaining readings will disappear as the reflected photons collide with a brick wall or whatever. Heisenberg's uncertainty principle aside, I don't want to worry conceptually about the effects of my observation on the reading. At the same time, my brain will only take into account as many readings as it can or wants to process.

Let's see what this means for a piece of software. In the BirdWatch application, there is a client side component or subsystem that holds the application state inside an **atom**. This component knows how to handle incoming messages, such as new tweets, and how to alter its internal state after applying some business logic to the previous state while taking the incoming messages into consideration. And then, there are multiple UI components that render the current application state. They don't require write-access to the application state; instead, they only render what they can observe. At the same time, only the latest instance of such an immutable state snapshot ever needs to be taken into account, as all previous ones are stale information by the time there's a newer snapshot.

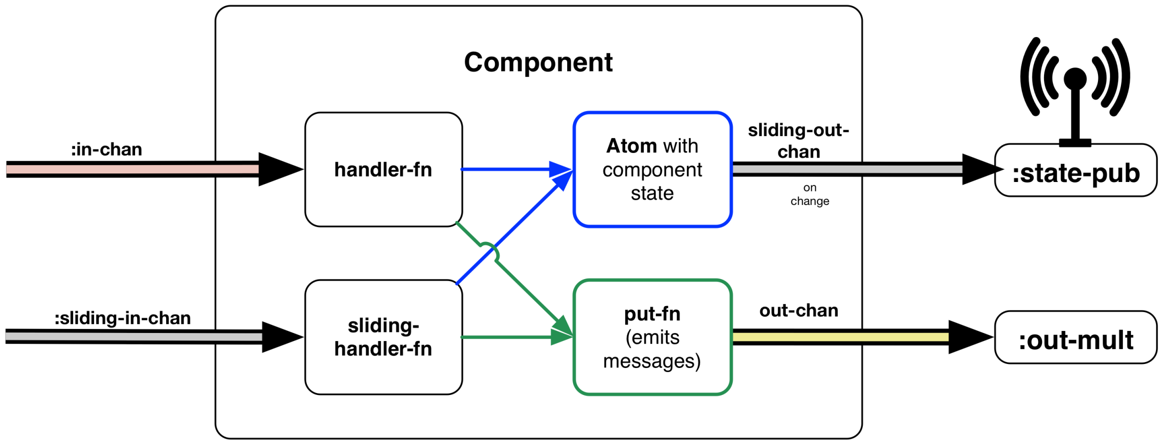
Looking at the BirdWatch application again, the `state component` receives new messages, including chunks with the last 5000 tweets or whatever you choose. The state component, which handles such chunk messages, mutates its internal state atom like 50-60,000 within less than 5 seconds after application startup. There's no reason to drain the battery by rendering every single one of the emitted snapshots into a virtual DOM representation and then have **React** detect the changes by diffing every such representation. Instead, all that needs to happen is rendering the newest snapshot 60 times per second to achieve a satisfactory user experience that appears to be running smoothly.

More broadly speaking, any component will have some internal state. This component will broadcast any mutation of its state atom by emitting an immutable snapshot of the latest state. Any consumer of such snapshots then only subscribes to as many as it can process, with the guarantee to always get the latest version whenever it is ready to consume a new snapshot. Core.async's **sliding buffer** is the building block that enables this behavior. It is a buffer of a defined size that will always make room for new messages by discarding the oldest unconsumed messages.

When dealing with state snapshots, we can even create one such buffer of *size one* that only holds the latest snapshot message. Components then emit snapshots of their internal state, without blocking unnecessarily when unknown recipients aren't ready for consumption. Such a mechanism is not something I should have to rewrite every single time; rather, the library should come with an abstraction for building such components, without writing repetitive boilerplate.

Potential observers can then subscribe to such state snapshots, all while again being guarded by another such sliding buffer against becoming overly busy with processing outdated snapshots.

Let's look at a possible design of a subsystem. Each component can, first of all, keep its state in an `atom`. The specifics of the initial state differ, so this is something we will have to pass to a component/subsystem "factory" function. The component will then automatically publish changes to this atom on a channel in the form of snapshots. Then, there's another outgoing channel for messages that we intend another component to process. Components do not have to interact with this channel directly. Instead, we can pass the component a function named `put-fn` that it can then call whenever it has a message to emit.



Subsystem/Component

On the input side of a component, we once again have two different channels. One of them is for handling messages that require processing every time. The other is for observing messages such as state snapshots, where only ever the last version is of interest. In either case, the existence of such channels as a means of taking information in is common to all components, whatever their purpose. For the differences in behavior, we only need to specify handler functions for the different message types. These handlers take a map with the state `atom` and the `put-fn` (among others) so they can change the component state and emit messages when required. Giving those handlers access to the application state is safe thanks to **Software Transactional Memory**, in case you worried about concurrency just now.

On the output side, there are the channels mentioned above, one for sending messages with some intent, and another one for emitting state snapshots. Publishing such snapshots is not something you have to worry about, it is taken care of by the library whenever any of your handlers updates the component state atom.

One important thing to note here is that the provided mechanisms for creating components work equally well on the Clojure/JVM and the ClojureScript/Browser side. In fact, the code is the same, it is written in `.cljc` files making use of **reader conditionals** that come with Clojure 1.7. In short, this feature allows maintaining a single code base for all platforms, where only the differences between platforms have to be implemented differently. I recommend reading the announcement linked above if you'd like to find out more about this new language feature.

The sweet thing about exploring and writing the **systems-toolbox** library is that it's not a mere thought experiment but also something that we're building a commercial system upon at my current gig. There, such subsystems are communicating with each other, connected over fast WebSockets, should messages need to traverse platform boundaries between client and server. Also, the latest version of the **BirdWatch** application uses this library both on client and server, providing a unified way of writing code on both sides. Also, all the UI components are built on top of this library, as we shall see.

Okay, so much about the motivation behind the library. Let's look at a simple example next.