How WhatsApp Works

WhatsApp or most of the other messaging apps rarely work on a peer to peer basis. So it wouldn't open a connection (from your device) to each of your friends' devices. Instead your device connects to their server. It could then use a custom TCP protocol or maybe HTTP to communicate your messages to the server. The server in return would dispatch them to your friends' devices. If your friend had their app open or at least the Sapp process running there might be a live connection to the server. WhatsApp will use that connection to send them your messages. If their app is "offline" then they might choose to send them a push notification instead.

Let's assume that Alice and Bob are friends on WhatsApp. So there is an an Alice actor and a Bob actor.

Let's trace a series of messages flowing back and forth:

1. Alice decides to message Bob. Alice's phone establishes a connection to the WhatsApp server and it is established that this connection is definitely from Alice's phone. Alice now sends via TCP the following message: "For Bob: A giant monster is attacking the Golden Gate Bridge". One of the WhatsApp front end server de-serializes this message and delivers this message to the actor called Alice.
2. Alice the actor decides to serialize this and store it in a file called "Alice's Sent Messages", stored on a replicated file system to prevent data loss due to unpredictable monster rampage. Alice the actor then decides to forward this message to Bob the actor by passing it a message "Msg1 from Alice: A giant monster is attacking the Golden Gate Bridge". Alice the actor can retry with exponential back-off till Bob the actor acknowledges receiving the message.
3. Bob the actor eventually receives the message from (2) and decides to store this message in a file called "Bob's Inbox". Once it has stored this message durably Bob the actor will acknowledge receiving the message by sending Alice the actor a message of its own saying "I received Msg1". Alice the actor can now stop it's retry efforts. Bob the actor then checks to see if Bob's phone has an active connection to the server. It does and so Bob the actor streams this message to the device via TCP.
4. Bob sees this message and replies with “For Alice: Let's create giant robots to fight them". This is now received by Bob the actor as outlined in Step 1. Bob the actor then repeats Step 2 and 3 to make sure Alice eventually receives the idea that will save mankind

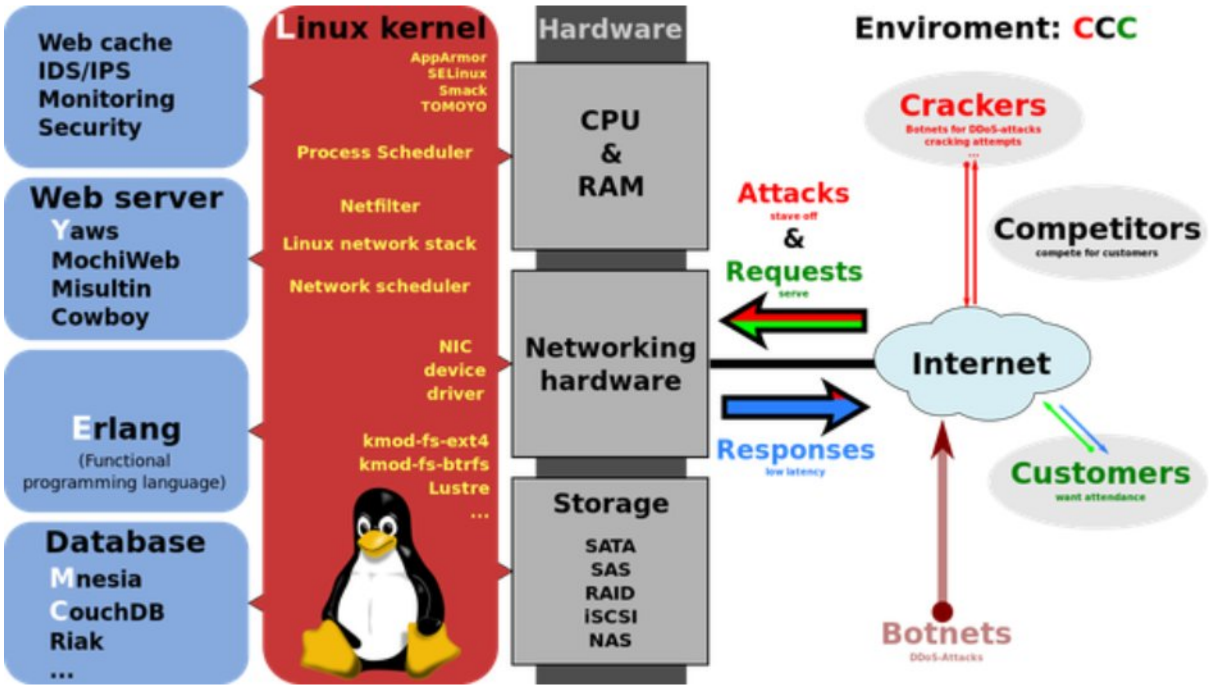
How does the registration process work internally in WhatsApp

WhatsApp used to create a username/password based on the phone IMEI number. This was changed recently. WhatsApp now uses a general request from the app to send a unique 5 digit PIN. WhatsApp will then send a SMS to the indicated phone number (this means the WhatsApp client no longer needs to run on the same phone). Based on the pin number the app then request a unique key from WhatsApp. This key is used as "password" for all future calls. (This "permanent" key is stored on the device). This also means that registering a new device will invalidate the key on the old device.

What protocol is used in WhatsApp

SSL socket to the WhatsApp server pools. All messages are queued on the server until the client reconnects to retrieve the messages. The successful retrieval of a message is sent back to the WhatsApp server which forwards this status back to the original sender (which will see that as a "checkmark" icon next to the message). Messages are wiped from the server memory as soon as the client has accepted the message

WhatsApp Architecture



WhatsApp actually uses the XMPP protocol instead of the vastly superior protocol that I outlined above, but you get the point. WhatsApp has chosen Erlang a language built for writing scalable applications that are designed to withstand errors. Erlang uses an abstraction called the Actor model for it's concurrency - http://en.wikipedia.org/wiki/Act.... Instead of the more traditional shared memory approach, actors communicate by sending each other messages. Actors unlike threads are designed to be lightweight. Actors could be on the same machine or on different machines and the message passing abstractions works for both. A simple implementation of WhatsApp could be:

Each user/device is represented as an actor. This actor is responsible for handling the inbox of the user, how it gets serialized to disk, the messages that the user sends and the messages that the user receives.

WhatsApp Stats

- 450 million active users, and reached that number faster than any other company in history.
- 32 engineers, one developer supports 14 million active users
- 50 billion messages every day across seven platforms (inbound + outbound)
- 1+ million people sign up every day
- $0 invested in advertising
- $60 million investment from Sequoia Capital; $3.4 billion is the amount Sequoia will make
- 35% is how much of Facebook's cash is being used for the deal
- Hundreds of nodes
- >8000 cores
- Hundreds of terabytes of RAM
- >70M Erlang messages per second
- In 2011 WhatsApp achieved 1 million established tcp sessions on a single machine with memory and cpu to spare. In 2012 that was pushed to over 2 million tcp connections. In 2013 WhatsApp tweeted out: On Dec 31st we had a new record day: 7B msgs inbound, 11B msgs outbound = 18 billion total messages processed in one day! Happy 2013!!!

WhatsApp Platform

Backend

- Erlang
- FreeBSD
- Yaws, lighttpd
- PHP
- Custom patches to BEAM (BEAM is like Java's JVM, but for Erlang)
- Custom XMPP

Frontend

- Seven client platforms: iPhone, Android, Blackberry, Nokia Symbian S60, Nokia S40, Windows Phone
- SQLite

Hardware

- Standard user facing server:
    - Dual Westmere Hex-core (24 logical CPUs);
    - 100GB RAM, SSD;
    - Dual NIC (public user-facing network, private back-end/distribution)

WhatsApp Product Goals

- Focus is on messaging. Connecting people all over the world, regardless of where they are in the world, without having to pay a lot of money. Founder Jan Koum remembers how difficult it was in 1992 to connect to family all over the world.
- Privacy. Shaped by Jan Koum's experiences growing up in the Ukraine, where nothing was private. Messages are not stored on servers; chat history is not stored; goal is to know as little about users as possible; your name and your gender are not known; chat history is only on your phone.

- WhatsApp server is almost completely implemented in Erlang.
    - Server systems that do the backend message routing are done in Erlang.
    - Great achievement is that the number of active users is managed with a really small server footprint. Team consensus is that it is largely because of Erlang.
    - Interesting to note Facebook Chat was written in Erlang in 2009, but they went away from it because it was hard to find qualified programmers.
- WhatsApp server has started from ejabberd
    - Ejabberd is a famous open source Jabber server written in Erlang.
    - Originally chosen because its open, had great reviews by developers, ease of start and the promise of Erlang's long term suitability for large communication system.
    - The next few years were spent re-writing and modifying quite a few parts of ejabberd, including switching from XMPP to internally developed protocol, restructuring the code base and redesigning some core components, and making lots of important modifications to Erlang VM to optimize server performance.
- To handle 50 billion messages a day the focus is on making a reliable system that works. Monetization is something to look at later, it's far far down the road.
- A primary gauge of system health is message queue length. The message queue length of all the processes on a node is constantly monitored and an alert is sent out if they accumulate backlog beyond a preset threshold. If one or more

processes falls behind that is alerted on, which gives a pointer to the next bottleneck to attack.

- Multimedia messages are sent by uploading the image, audio or video to be sent to an HTTP server and then sending a link to the content along with its Base64 encoded thumbnail (if applicable).
- Some code is usually pushed every day. Often, it's multiple times a day, though in general peak traffic times are avoided. Erlang helps being aggressive in getting fixes and features into production. Hot-loading means updates can be pushed without restarts or traffic shifting. Mistakes can usually be undone very quickly, again by hot-loading. Systems tend to be much more loosely-coupled which makes it very easy to roll changes out incrementally.
- Google's push service is used on Android.
- More users on Android. Android is more enjoyable to work with. Developers are able to prototype a feature and push it out to hundreds of millions of users overnight, if there's an issue it can be fixed quickly. iOS, not so much.
- Experienced lots of user growth, which is a good problem to have, but it also means having to spend money buying more hardware and increased operational complexity of managing all those machines.
- Need to plan for bumps in traffic. Examples are soccer games and earthquakes in Spain or Mexico. These happen near peak traffic loads, so there needs to be enough spare capacity to handle peaks + bumps. A recent soccer match generated a 35% spike in outbound message rate right at the daily peak.
- Initial server loading was 200 simultaneous connections per server.

    - Extrapolated out would mean a lot of servers with the hoped for growth pattern.
    - Servers were brittle in the face of burst loads. Network glitches and other problems would occur. Needed to decouple components so things weren't so brittle at high capacity.
    - Goal was a million connections per server. An ambitious goal given at the time they were running at 200K connections. Running servers with headroom to allow for world events, hardware failures, and other types of glitches would require enough resilience to handle the high usage levels and failures.

Tools and Techniques Used to Increase Scalability

- Wrote system activity reporter tool (wsar):
    - Record system stats across the system, including OS stats, hardware stats, BEAM stats. It was build so it was easy to plugin metrics from other

    systems, like virtual memory. Track CPU utilization, overall utilization, user time, system time, interrupt time, context switches, system calls, traps, packets sent/received, total count of messages in queues across all processes, busy port events, traffic rate, bytes in/out, scheduling stats, garbage collection stats, words collected, etc.
  - Initially ran once a minute. As the systems were driven harder one second polling resolution was required because events that happened in the space if a minute were invisible. Really fine grained stats to see how everything is performing.
- Hardware performance counters in CPU (pmcstat):
  - See where the CPU is at as a percentage of time. Can tell how much time is being spent executing the emulator loop. In their case it is 16% which tells them that only 16% is executing emulated code so even if you were able to remove all the execution time of all the Erlang code it would only save 16% out of the total runtime. This implies you should focus in other areas to improve efficiency of the system.
- dtrace, kernel lock-counting, fprof
  - Dtrace was mostly for debugging, not performance.
  - Patched BEAM on FreeBSD to include CPU time stamp.
  - Wrote scripts to create an aggregated view of across all processes to see where routines are spending all the time.
  - Biggest win was compiling the emulator with lock counting turned on.
- Some Issues:
  - Earlier on saw more time spent in the garbage collections routines that was brought down.
  - Saw some issues with the networking stack that was tuned away.
  - Most issues were with lock contention in the emulator which shows strongly in the output of the lock counting.
- Measurement:
  - Synthetic workloads, which means generating traffic from your own test scripts, is of little value for tuning user facing systems at extreme scale.
    - Worked well for simple interfaces like a user table, generating inserts and reads as quickly as possible.
    - If supporting a million connections on a server it would take 30 hosts to open enough IP ports to generate enough connections to test just one server. For two million servers that would take 60 hosts. Just difficult to generate that kind of scale.
    - The type of traffic that is seen during production is difficult to generate. Can guess at a normal workload, but in actuality see

        networking events, world events, since multi-platform see varying behavior between clients, and varying by country.
  - Tee’d workload:
    - Take normal production traffic and pipe it off to a separate system.
    - Very useful for systems for which side effects could be constrained. Don’t want to tee traffic and do things that would affect the permanent state of a user or result in multiple messages going to users.
    - Erlang supports hot loading, so could be under a full production load, have an idea, compile, load the change as the program is running and instantly see if that change is better or worse.
    - Added knobs to change production load dynamically and see how it would affect performance. Would be tailing the sar output looking at things like CPU usage, VM utilization, listen queue overflows, and turn knobs to see how the system reacted.
  - True production loads:
    - Ultimate test. Doing both input work and output work.
    - Put server in DNS a couple of times so it would get double or triple the normal traffic. Creates issues with TTLs because clients don’t respect DNS TTLs and there’s a delay, so can’t quickly react to getting more traffic than can be dealt with.
    - IPFW. Forward traffic from one server to another so could give a host exactly the number of desired client connections. A bug caused a kernel panic so that didn't work very well.
- Results:
  - Started at 200K simultaneous connections per server.
  - First bottleneck showed up at 425K. System ran into a lot of contention. Work stopped. Instrumented the scheduler to measure how much useful work is being done, or sleeping, or spinning. Under load it started to hit sleeping locks so 35-45% CPU was being used across the system but the schedulers are at 95% utilization.
  - The first round of fixes got to over a million connections.
    - VM usage is at 76%. CPU is at 73%. BEAM emulator running at 45% utilization, which matches closely to user percentage, which is good because the emulator runs as user.
    - Ordinarily CPU utilization isn’t a good measure of how busy a system is because the scheduler uses CPU.
  - A month later tackling bottlenecks 2 million connections per server was achieved.

        - BEAM utilization at 80%, close to where FreeBSD might start paging. CPU is about the same, with double the connections. Scheduler is hitting contention, but running pretty well.
    - Seemed like a good place to stop so started profiling Erlang code.
        - Originally had two Erlang processes per connection. Cut that to one.
        - Did some things with timers.
    - Peaked at 2.8M connections per server
        - 571k pkts/sec, >200k dist msgs/sec
        - Made some memory optimizations so VM load was down to 70%.
    - Tried 3 million connections, but failed.
        - See long message queues when the system is in trouble. Either a single message queue or a sum of message queues.
        - Added to BEAM instrumentation on message queue stats per process. How many messages are being sent/received, how fast.
        - Sampling every 10 seconds, could see a process had 600K messages in its message queue with a dequeue rate of 40K with a delay of 15 seconds. Projected drain time was 41 seconds.
- Findings:
    - Erlang + BEAM + their fixes - has awesome SMP scalability. Nearly linear scalability. Remarkable. On a 24-way box can run the system with 85% CPU utilization and it's keeping up running a production load. It can run like this all day.
        - Testament to Erlang's program model.
        - The longer a server has been up it will accumulate long running connections that are mostly idle so it can handle more connections because these connections are not as busy per connection.
    - Contention was biggest issue.
        - Some fixes were in their Erlang code to reduce BEAM's contention issues.
        - Some patched to BEAM.
        - Partitioning workload so work didn't have to cross processors a lot.
        - Time-of-day lock. Every time a message is delivered from a port it looks to update the time-of-day which is a single lock across all schedulers which means all CPUs are hitting one lock.
        - Optimized use of timer wheels. Removed bif timer
        - Check IO time table grows arithmetically. Created VM thrashing has the hash table would be reallocated at various points. Improved to use geometric allocation of the table.
        - Added write file that takes a port that you already have open to reduce port contention.

    - Mseg allocation is single point of contention across all allocators. Make per scheduler.
    - Lots of port transactions when accepting a connection. Set option to reduce expensive port interactions.
    - When message queue backlogs became large garbage collection would destabilize the system. So pause GC until the queues shrunk.
  - Avoiding some common things that come at a price.
    - Backported a TSE time counter from FreeBSD 9 to 8. It's a cheaper to read timer. Fast to get time of day, less expensive than going to a chip.
    - Backported igp network driver from FreeBSD 9 because having issue with multiple queue on NICs locking up.
    - Increase number of files and sockets.
    - Pmcstat showed a lot of time was spent looking up PCBs in the network stack. So bumped up the size of the hash table to make lookups faster.
  - BEAM Patches
    - Previously mentioned instrumentation patches. Instrument scheduler to get utilization information, statistics for message queues, number of sleeps, send rates, message counts, etc. Can be done in Erlang code with procinfo, but with a million connections it's very slow.
    - Stats collection is very efficient to gather so they can be run in production.
    - Stats kept at 3 different decay intervals: 1, 10, 100 second intervals. Allows seeing issues over time.
    - Make lock counting work for larger async thread counts.
    - Added debug options to debug lock counters.
  - Tuning
    - Set the scheduler wake up threshold to low because schedulers would go to sleep and would never wake up.
    - Prefer mseg allocators over malloc.
    - Have an allocator per instance per scheduler.
    - Configure carrier sizes start out big and get bigger. Causes FreeBSD to use super pages. Reduced TLB thrash rate and improves throughput for the same CPU.
    - Run BEAM at real-time priority so that other things like cron jobs don't interrupt schedule. Prevents glitches that would cause backlogs of important user traffic.
    - Patch to dial down spin counts so the scheduler wouldn't spin.

    - Mnesia
        - Prefer os:timestamp to erlang:now.
        - Using no transactions, but with remote replication ran into a backlog. Parallelized replication for each table to increase throughput.
    - There are actually lots more changes that were made.

<u>Things to consider:</u>

1. You might not have control over clients sending GPS coordinates to the server every 10 minutes. If your client is running on a mobile device, the OS might decide to starve you from resources or just kill your process.
2. You need to maintain state for clients that are connected to your server to make sure you can send messages to active clients when your requirements are met. This is a slight modification of the stock "Comet app" example that almost every framework has.
3. Establishing a TCP connection is not a very big waste of resources either from the client's side or from the server's side. If your server software ecosystem supports non-blocking IO, the state required per connection is tiny. You could support upwards of 100k connections on a mediocre box if you tried hard. If you are on the JVM Netty might help you here. Python has Twisted and Tornado. C++/ C can make use of epoll, kqueue or select if you are on a *NIX system. Golang supports a high number of connections through its standard library. We have addressed vertical scalability here i.e. how many users could you support on a simple box.If you really want to scale out and build a distributed system that maintains state, you might want to consider Erlang (with OTP) or other implementations of the Actor model, like Akka (JVM) which also supports remote messages. A combination of event sourcing and a message passing architecture could get you all horizontal scalability you need.