NTNU
Det skapende universitet

Distribuert database for posisjonslagring

Eirik Alderslyst Nygaard

Master i datateknikk
Oppgaven levert: Juni 2010
Hovedveileder: Svein-Olaf Hvasshovd, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

Denne oppgaven vil gå ut på å designe et system for å holde oversikt over plasseringen til person ut fra lokasjonsdata som GPS eller mobilmaster. Det vil være behov for å designe en database som håndterer store mengder oppdateringer av lokasjonsinformasjonen til hver enkelt bruker samt å sende ut denne dataen til de som abonnerer på plasseringen til den oppdaterte brukeren.

Systemet som skal lages må være redundant så noder kan kræsje uten at det fører til nedetid i systemet. Noen muligheter vil da være en hot standby for alle noder, eller at nabonoder kan dele på brukerene som var på noden som gikk ned.

Brukerene vil sende kontinuerlige oppdateringer med hvor de har forflyttet seg til systemet, og disse må håndteres fortløpende. Det er derfor nødvendig å se på hvordan systemet skal håndtere at det kommer flere oppdateringsforespørsler enn det kan takle, og passe på at det alltid er ressurser nok tilgjengelig for å sende ut oppdateringene til de som ønsker denne.

For å håndtere dette vil den distribuerte databasen være geografisk spredt for at de som sender oppdateringer alltid vil være i nærheten til en server for å minke latensen mellom de, og det kan være fordelaktig at en bruker å migrere mellom servere ut fra hvor i verden de er for øyeblikket. For at en server skal kunne sende ut oppdatert informasjon til de som vil vite hvor en person er må det være en mekanisme i nettverket for å effektivt sende ut denne informasjonen til riktige mottakere.

Oppgaven gitt: 15. januar 2010
Hovedveileder: Svein-Olaf Hvasshovd, IDI

Abstract

Social interaction is a integral part of today's society. And people would often like to know where their friends are. In this thesis a system will be described that can receive location updates from people and relay them to any interested party.

The system is distributed to make sure it will scale up and handle a huge amount of users. And geographically spread out and a user will always connect to the closest set of server so that the latency during communication is kept to a minimum. As well as keeping the latency low this also allows temporary failures in in global communication without denying the user the ability to send updates.

All communication which will be over longer distances are done by the servers which will most likely have better bandwidth and connection to the rest of the world.

Finally all users will belong to one geographical zone which is responsible for knowing where the user is and having the most recent subscriber information.

Contents

1 Introduction **1**

2 Background **3**
2.1 Key value stores . . . 3
2.1.1 Redundancy . . . 3
2.1.2 Consistency . . . 4
2.2 GSM . . . 4
2.2.1 Cells . . . 5
2.2.2 Handover . . . 5

3 State of the art **7**
3.1 Google Latitude . . . 7
3.2 mBuddy . . . 7
3.3 Delivery services . . . 8

4 Architecture **9**
4.1 Zone . . . 9
4.1.1 Home zone . . . 11
4.1.2 Redundancy . . . 11
4.2 Consistency . . . 11
4.3 Migration between zones . . . 12
4.3.1 Special cases . . . 12
4.4 Database layout . . . 14
4.5 User id . . . 15
4.6 User lookup . . . 16
4.7 Subscribing for user updates . . . 16
4.8 Update notification . . . 16
4.9 Geographical location . . . 16
4.10 Home zone selection . . . 17
4.11 Network protocol . . . 17
4.12 Server discovery . . . 17
4.13 Server overload . . . 17
4.14 Use cases . . . 18
4.14.1 New user . . . 18

4.14.2 Subscribe to updates from a single user . . . 18
4.14.3 User update with only local subscribers . . . 18
4.14.4 User update with foreign subscribers . . . 19
4.14.5 User update in foreign zone . . . 19
4.14.6 Home zone server downtime . . . 19
4.14.7 Waiting for user updates . . . 19
4.14.8 Migrating from a home zone . . . 20
4.14.9 Migrating to a home zone . . . 20
4.14.10 Back and forth migration . . . 20
4.14.11 Entire zone downtime . . . 20

5 Discussion **21**
5.1 Multiple zones . . . 21
5.1.1 Zone failure . . . 21
5.2 Mapping location to zone . . . 22
5.3 Database layout . . . 22
5.4 User id . . . 23
5.5 Home zone . . . 23
5.6 Subscription . . . 24
5.7 Network protocol . . . 24
5.7.1 Transmission Control Protocol . . . 24
5.7.2 User Datagram Protocol . . . 25
5.7.3 Stream Control Transmission Protocol . . . 25
5.7.4 Comparison . . . 25
5.8 Full mesh communication . . . 26
5.9 Server overload . . . 26

6 Implementation **27**
6.1 Storage . . . 27
6.2 Protocol . . . 27
6.2.1 Commands . . . 28
6.2.2 Parsing . . . 29
6.3 Request queue . . . 30
6.4 Threading model . . . 30
6.5 Network communication . . . 31

7 Future work **33**
7.1 Find nearby friends . . . 33
7.2 Authorization and authenticating . . . 33
7.3 Outlier detection and correction . . . 33
7.4 Keep historic data . . . 34

8 Conclusion **35**

List of Figures

4.1 Zones overview 10
4.2 User migrating from zone A to zone B and then back to zone A . . . 13
4.3 User migrating continuously between zone A and zone B 14

6.1 Update user command example 28
6.2 Fetch command example 28
6.3 Subscribe command example 28
6.4 Subscribers command example 29
6.5 Poll command example 29
6.6 New user command example 29

List of Tables

4.1 User location database layout for home zone 15
4.2 User location database layout for visiting zone 15
4.3 Subscriber database layout 15
4.4 Subscriber of visitor database layout 16
4.5 Current zone 16

Chapter 1

Introduction

This thesis describes a system for storing and distributing the location of a huge amount of individuals or items (from now on known as users). The system wait for updates from the objects that move around, and in real time push this information to the interested parties, as well as store it.

To maintain as high uptime as possible, and remain scalable, the system is split into several servers which communicate. To handle server downtime there exist servers that are clones of each other, and internally keep them self up to date. Linear scaling must be supports to allow new users to be created without degrading the quality of the rest of the system, by doubling the hardware capacity the number of simultaneously handled users should also double.

Since every user can continuously update the server with their location, a server might end up getting too much traffic. Handling server overload is therefor an important aspect to keep the system running smoothly.

To minimize the latency for update requests, a geographical dispersion of the nodes in the system is needed. With a short geographical distance the latency will be lower than if the server was located far away. By physically placing server different places, areas with a dense user base can have servers that cover a smaller area, while servers can cover a greater area if the user base is rather sparse.

This system can be in several ways, either for people to keep track of where theirs friends are, for researchers to keep track of wild animals, or the postal service or other delivery businesses.

A person might want to find out where a friend is that he is supposed to meet, or find a friend nearby which he could do something with. Having location tracking can also be used to ind other people you don't already know with similar interests if the system is connected with a database which contains interests.

Using it for locating people will also allow people with access to the data to match that and other available person data to find possible friends of yours, or the other way around, by matching your interests and location find other people you share interests with in your area.

A system which always know where someone is located can be a privacy problem. People does not always want the world to know where they are. Or there

might be some people you would not want to know where you are.

Farmers can use such a system to always have an up to date knowledge of where their animals are. By matching this data up against the trackers wild animals is marked with, one could avoid attacks by herding their animals to safer places, or sending out a watchman when the two animal groups are know to frequent in the same area.

A distributed design can leverage recent advances in key-value database in the later years. Specially their algorithms for keeping multiple nodes in sync and load balancing.

The system itself is described in chapter 4. The architecture defines interaction between servers and clients, as well as communication between two servers or zones. In chapter 5 other possible solutions to the architecture is talked about, and why the architecture was chosen.

The test implementation of a server which handles clients are then described in chapter 6, this also contains the protocol used to communicate. And the conclusion sums up the entire system in chapter 8.

Chapter 2

Background

2.1 Key value stores

Key value stores are a generalized database for storing and retrieving data based on a given key. The key value stores are designed to scale up to massive amounts of data[1], and normally includes built in data replication support to keep the same data stored on several servers. Handling a huge amount of read, writes and keep data always available requires a network of servers which can all answer database requests.

To allow for massive amounts of information to be stored, the data is spread out across several servers. One of the solutions for selecting which server get which data is to use consistent hashing[2]. Using consistent hashing servers can be added and removed without the need to relocate data.

Both keys and values can be seen as an object consisting of several other objects, or as a single element. An element can be integer, decimal number or strings. Storing data is usually called a `put` operation, and retrieving is done by a `get` operation.

Key value stores are used by companies like Facebook[1] and Amazon[2] to handle their massive amount of users[3]. When there are many database servers in a setup one must assume one will always fail, and the key value stores are designed for the failure case. No data should become unavailable if a server goes down, and it should still be possible to store new data. Even if the used key would normally end up on the failed server.

2.1.1 Redundancy

To achieve durability of the stored data it must be replicated to several servers in the key value store network. The replication also increases the availability because

[1] http://www.facebook.com

[2] http://www.amazon.com

[3] Facebook had 400 million active users in May 2010[3], and half of them used the web page each day

even if the server that originally got the store request it will be possible to retrieve the object from another server.

When a server receives a new key value pair, it will be stored and which other servers that in addition will get a copy is decided based on the key and to which other servers the original server is connected.

Replication causes some trouble when it comes to consistency of the data which is explained in subsection 2.1.2, but the negative impact is not problematic enough to change from a loosely connected set of servers which work together to form a redundant and high-availability database.

2.1.2 Consistency

Updating a nested structure like an object can result in inconsistent reads when in the middle of writing a new object. Transactions is a normal way to deal with this, but are not always available in common key value stores[4]. These databases rely on atomic operations, and requires that the programmer make sure that inconsistencies does not occur.

Another problem with consistency in key value stores is the fetch-store cycle for updating. Since a series of operations are not guaranteed to be atomic two clients could first fetch an object. Then after the old version has been retrieved by both they update separate parts of the object and write the changes back. One way to handle this is for the database server to attach a serial number to every object it sends to a client, and when the client sends an update back to the server the serial must match the serial for the object. If the serials does not match the update is denied. This ensures that the object was not updated by someone else in between the fetch and update requests from the client.

When data is replicated to several servers, and an update to a object can be done to any one of those servers the client can not handle a conflict when doing the update. The time it takes the object to be updated on several server to achieve redundancy allows two clients to update the same object on different servers at the same time without knowing of the other persons update. Or when a server becomes unavailable without being able to send out the updated object to the other servers. Instead of resolving the conflict when doing the update, the conflict can be handled when the client reads the object.

2.2 GSM

GSM[5] is a global mobile communication system which allows people to communicate with speech in real time without being connected by any wires. There were some problems when different countries developed their own systems. But a standardization effort helped consolidate all into one connected network. Cell towers are used to communicate with the wireless devices, and relays the signal so it ends up at the correct endpoint.

2.2.1 Cells

GSM networks are cells of overlapping cell towers. These cell towers are responsible for the wireless communication within their area, and relaying the data it receives to the correct next step. An overlap exists between the different cells to allow devices to travel between the cells without losing connectivity.

The cell towers are planned with enough range so the overlap is enough to allow handovers to be performed properly, but small enough to minimize the interference between the towers.

2.2.2 Handover

The handover protocol is a way to allow a device to travel between two cells without losing open connections. A handover can also be initiated if the channel a device is communicating on is overloaded and the device needs to switch to a different channel.

Handovers are considered when the signal strength drops below a given threshold. This could happen for two reasons. Either the device has left the current cell, or it does not supply enough power to the sender. A power increase may be tried before a handover is initiated.

Chapter 3

State of the art

For location tracking there are several services that are in use today. Google has a system closest to the system described in this paper, but no technical information is available for it except for the API used to communicate with it. Several mobile phone service providers allows you to track your friends using their GSM location if they allow you to follow them. Telenor is said to have done research for a location system, but no information for this is publicly available. Most package delivery services allows you to see where your package is currently located, but it has several short comings.

Even with these system available not much information is available to look at how the systems operate.

3.1 Google Latitude

Latitude from Google is a social location network. You can tell your friends where you are. It has integration with Google Maps, and allows you to find friends located in close proximity to yourself. It also records your location history for plotting on a map, and allows you to see where you were at a given time.

The user is self responsible for sending out location information to Latitude. It can either be done by adding an application to your smart-phone which automatically sends updates, or by using the Latitude API.

The API gives you access to your location history, your location and the ability to update you current location. This allows users to write new applications which can use location information, and not be locked into the application Google publishes for the service.

3.2 mBuddy

This system uses the GSM network to get the position of a cell phone. Most people always carry around their cell phone, and can therefor be tracked continuously. The major problem with using the GSM[6] network is that it is not very accurate. The

accuracy depends on how many cell phone towers the cell phone is connected to, and how far apart they are. In places where cell phone towers are located close to each other (like in a big city), you will be able to tell where a person is within a few hundred meters[7, Support]. However when people are out hiking in the mountains or forests the data collected will only be able to pinpoint a person within several kilometers. The accuracy of the location also depends on the base station you are connected to. Normally you would be connected to the closest base station which would give good accuracy, but if that base station is overloaded you might be transferred to one further away which would decrease the accuracy.

3.3 Delivery services

Several delivery services support package tracking. The Norwegian postal service, FedEx, and DHL are a few examples. These tracking systems does however not operate in real time. Instead they rely on packages to be registered at each location they arrive.

This results in packages that does not get updated at each location, either because it did not get properly scanned, or it was sent through to the next location without passing a scanner.

These delays in location updates can be annoying. The delivery company might end up not knowing where a package is for a time which leads to harder shipping planning.

Chapter 4

Architecture

This system will support storing the latest geographical position of users, and allow others to subscribe to users and be notified when their location is updated. The architecture will be fully redundant and downtime for a set of nodes will not influence the availability of the rest of the system.

The database will consist of several zones, where each zone is responsible for a geographical area. A user will always communicate with the zone that covers the area he is located within. Each user will also be associated with a home zone which has control over that user.

Migration of a user will handle the case where a user travels from one geographical zone to another. Control has to be transferred from the current zone to the next and subscribers to the user's location has to be transmitted.

Figure 4.1 shows a set of three zones and how they communicate. The network is a full mesh where every zone will communicate directly with the other zones when that is required.

Each zone contains four tables of data, users, subscribers, visitors and visitors subscribers. These are all described in section 4.4. User and subscribers are tables with data for the users which has the zone as their home zone, and visitors and visitors subscribers are a merged table with data from the home zones for the users that have temporarily migrated to this zone.

4.1 Zone

A zone contains a list of all the users which are located within the geographic area the zone is responsible for. Information about all the subscribers to a user is also stored in the zone the user is located. This way updated can be sent out directly without looking up subscribers.

Each zone has its own unique identifier to separate them from each other.

Consider history data, how do we want to do that and where should it be stored, transferred to the home zone when a migration occurs?

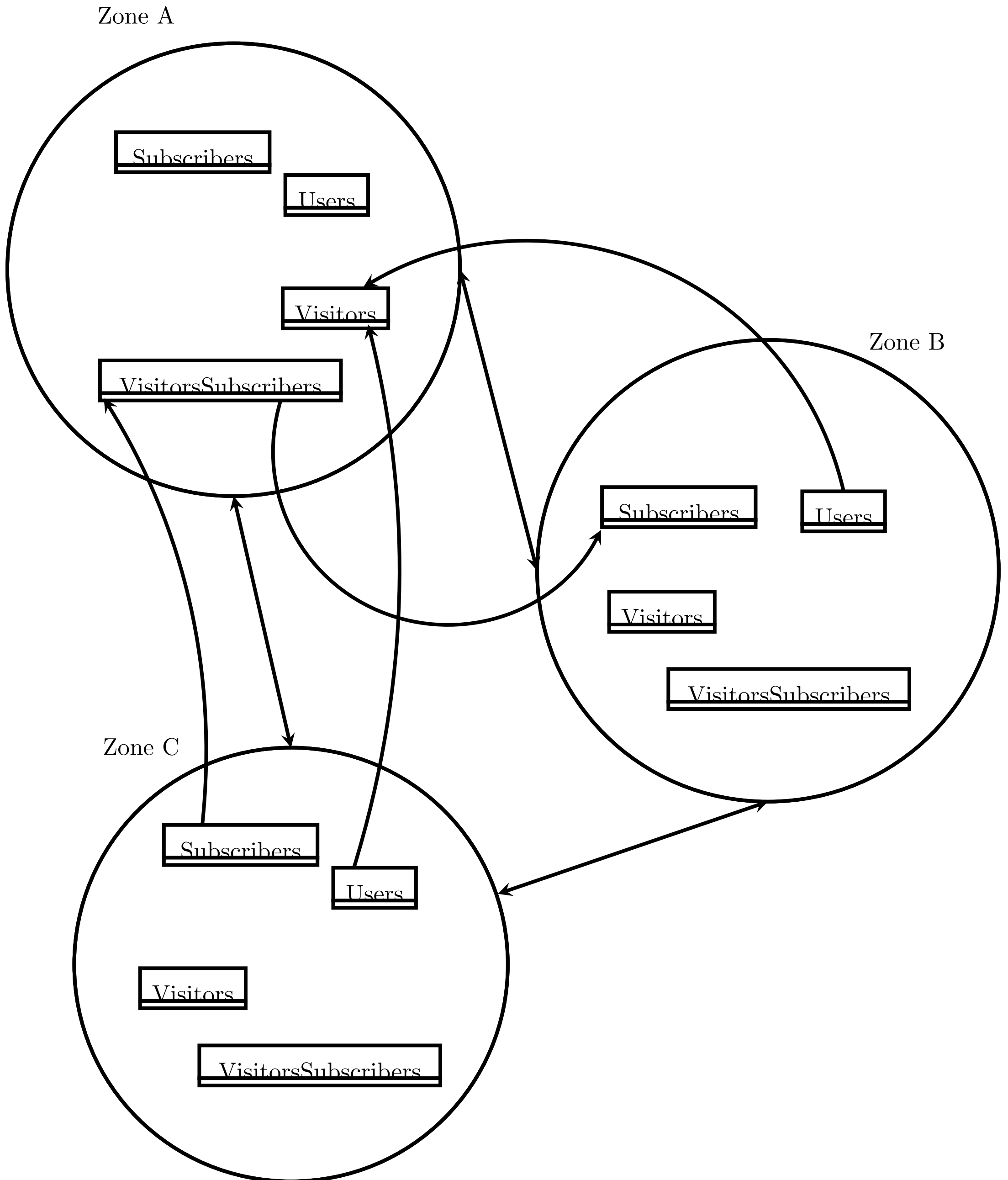

Figure 4.1: Zones overview

4.1.1 Home zone

Each home zone is responsible for keeping control of the users related to the zone. The home zone can allow or deny the migration of a user, and will always hold the most recent subscriber information. The home zone will be chosen by the user itself depending on where he is when signing up, and where his home location is.

4.1.2 Redundancy

Only the home zone data will be duplicated for redundancy. The subscribers of user and the user location databases for visiting user can always be recreated on demand from the data in the home zone. Keeping several copies of the data allows a server to crash without degrading the quality of a zone. If the data does not have several copies a crash of one server will result in data loss, but when it is copied to several servers data will still be available even if a server crashes.

Data from visiting users are only stored on the current active server within a zone to avoid duplicating data more than is needed. Since the data is already stored with redundancy it can be retrieved again if it is lost on the zone the user is visiting. This avoids keeping all the servers within a zone synchronized when it handles visiting users.

When a zone gets a location update from a user it does not have the information for, it will send a migration request to the home zone of that user and the migration protocol will take over and handle the missing data as if the user just came from another zone.

Each zone is responsible for always keeping at least one up and running server. The servers within one zone should therefor not all be kept at the same location or use the same network connection.

4.2 Consistency

To guarantee that the latest update by a user is the one retrieved when asking for that user's location. The location is first requested from the zone the user has as his home zone. If the user is located within his home zone the location will be sent back, if not the zone the user is located within is sent as the response.

The requester can then request the user's location from that zone instead. If during that time the user has migrated to a new zone the process has to be started from the beginning.

Consistency is also a problem within the same zone, not only when communicating across zones. To make sure the latest location information is always sent out from a zone, the system assumes that each server in a zone will always be able to communicate with each other when they are online.

4.3 Migration between zones

Zone migration is supervised by the home zone of the migrating user. There are three different kinds of migrations. The first is when migrating from a user's home zone, the second when the user is migrating to his home zone, and the third when the user migrates between two zones where neither is his home zone.

Migrating from a home zone requires the home zone to send all the subscriber information for that user as well as the last location the user reported in. The zone the user migrated to is then stored at the home zone to make sure that requests to that user can be redirected to the correct zone without first having to do a search for that user.

Migrating back to a user's home zone requires less data being sent. The only data that need to be exchanged between the active zone and the home zone is the latest location update from the user. The zone the user migrates away from will then remove all the data relating to the migrating user, and the home zone for the user will set the new current zone for the user to be itself.

The home zone is always responsible when a user is migrating. So when migrating between two foreign zones is still controlled by the user's home zone. The zone the user migrates from will send the user's latest location to the home zone, which will pass it on to the zone the user is migrating too, together with the subscriber list for the user. And finally the current zone table will be updated with the zone the user migrated to.

A migration is not explicitly request by a user. Instead the zones will automatically handle migration when a user sends a request to a zone that is not currently his active zone.

If a migration is already ongoing, or there are temporary problems between zones a migration can be temporary denied. If that happens the user must continue to communicate with the previous zone until the migration is successful.

In case of the required data is not able to be transferred, the migration will be denied and the client must retry the migration at a later time.

4.3.1 Special cases

There are corner cases when it comes to user migration between zones, which can result in problems when the home zone thinks that the user is in one zone, when he actually is in another. These cases are only a problem for a short time until a full migration has taken place, but it might result in losing some data and must be handled properly.

Back and forth

When a user is only in a zone for a short time and then goes back to the zone he came from, as illustrated in Figure 4.2. The home zone will not be fully synchronized with the current state the user is in.

This is handled by never allowing a user to migrate between zones while a migration is active. The home zone for a user will always know if a migration

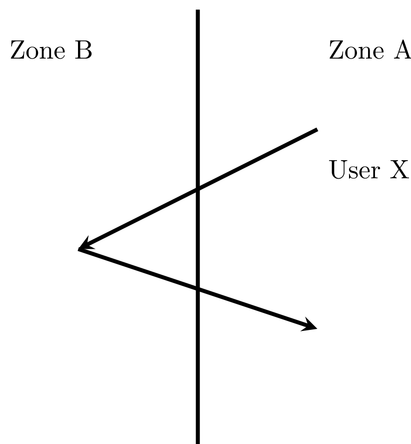

Figure 4.2: User migrating from zone A to zone B and then back to zone A

has started, and which steps are left before it is finished. Until all those steps are completed a new migration will always be denied, and the user must go back to the previous zone and send requests there instead.

Zig zag

When the user alternates between two zones continuously for an extended period of time, as illustrated in Figure 4.3, the home zone will "always" be out of sync.

Always changing between zones will result in too much wasted bandwidth, and unnecessary communication. When traveling along a border between two zones and constantly trying to migrate back and forth an exponential increase in the delay between each allowed migration is added. The first time it migrates it just have to wait for a full migration is done before it is allowed to migrate back, but the second time an extra delay is added. And if the user tries to migrate again before the delay has passed the time before the next allowed migration will be increased even more.

Entire zone downtime

A collapse of an entire zone can be detected by connecting to all servers in that zone and not being able to communicate with any of them. A user is not allowed to migrate if his home zone is down. Nor will it be possible to subscribe to a user which does not have a working home zone.

Any user with a home zone that is currently down will only be able to send new updates if they were located in another zone when their home zone went down. If a user was only visiting a zone that went down he can chose to migrate to one of the neighboring zones and use that for as long as the zone is down.

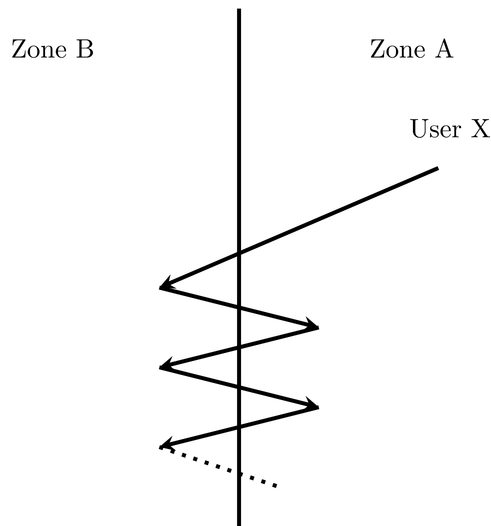

Figure 4.3: User migrating continuously between zone A and zone B

An entire zone should not go down if the servers in the zone is configured with proper redundancy. But some times multiple communication paths are not available, or several servers fail at the same time because of power outage or natural disasters.

4.4 Database layout

The database layouts are different for users located in their home zone and for those visiting a zone. This separation is there because the data is handled different ways. The user data for a visiting user will be added and removed much more frequently than the data for home zone user data, which will only be updated.

The table layout for the user data stored at the user's home zone is described in Table 4.1. **User id** is the primary key and denotes the unique user identificator across the entire system. The current location for the user is stored in **Physical location**, and the time of the last update is in **Last updated**. **User id** has the format described in section 4.5 and contains the home zone information for when that is required. **Physical location** and **Last updated** is only updated if the user is located in his home zone, or when the user location is requested from the user's current zone.

To efficiently send out updates to all the subscribers which want the location of a user, a list of subscribers is stored as seen in Table 4.3, the **User id** and **Subscriber id** combined is the primary key, since these identificators also contain zone information contacting the correct zone does not require any more lookups and requests can be relayed quickly.

User id	Physical location	Last updated

Table 4.1: User location database layout for home zone

User id	Physical location	Last updated

Table 4.2: User location database layout for visiting zone

When a user is visiting a zone a tuple with the information in Table 4.2 will be created or updated when new location information from the user is pushed to the system. In this table the **User id** is the primary key for the data, while **Physical location** is the actual location information and **Last updated** is the time of the latest location update.

To be able to send send out location updates to subscribers the subscriber table from the home zone is duplicated to avoid fetching it from the home zone each time, and to be able to relay the information directly without going back to the home zone.

The user which is visiting another zone will also have a record created in their home zone storing which zone they are currently in. This entry is shown in Table 4.5. The home zone must always know in which zone its users are located to be able to handle migration requests and send updates to the user without searching for the zone the user is currently in.

4.5 User id

The user id consists of the home zone server id and a serial number which is unique within the server where it was generated on. When the user id contains the home zone server id no discovery protocol is required to detect which zone a user belongs to. The format of the user id is `serverid-userid`.

Having a user id with two parts allows each server to hand out new unique user ids without synchronizing with the other server in the network. A user defined alias is not supported because. This is because it would have to be globally unique, and therefor require a synchronization between all the zones in the network, or a special centralized server to handle creation of them. against a centralized entity. A per server alias could possibly be allowed on the form `serverid-alias`, since it then would still contain the information to go directly to the correct server and it would not require any more work than checking for a unique user id to verify that there are not duplicate aliases.

User id	Subscriber id

Table 4.3: Subscriber database layout

User id	Subscriber id

Table 4.4: Subscriber of visitor database layout

User id	Visiting zone id

Table 4.5: Current zone

4.6 User lookup

Because a user id consists of both the home zone id and a unique serial, finding a user is done by parsing the zone id from the user id. And connect to that zone to verify that the user is valid.

4.7 Subscribing for user updates

A subscribe request is sent directly to the home zone of the user the subscriber wants to get updated information about. The subscriber is stored in the subscriber list and will from that point be notified of updates.

If a user that is being subscribed to is currently not located in his home zone the home zone is responsible for sending out the new subscriber data to the active zone for that user. The information is relayed instantly from the home zone to the active zone to make sure the change takes effect as soon as possible to minimize the delay between subscribing to a user and getting the first update.

4.8 Update notification

An update notification is a message sent out to all subscribers of a user that has updated his location. The notification contains which user did the update, the location of the user and the timestamp for the update.

Updates are required to be sent out in real time when new location information is pushed by a user. The subscriber list for the user is grouped by zone and updates are sent out in batch to each zone. Batch notifications are required to cut down on the bandwidth between zones.

4.9 Geographical location

The location is transmitted in the NAD83[8] format, with high precision floating point numbers it supports a very high accuracy on the pinpointing. This allows tracking even small packages located in small spaces if that is required.

Any device that gives the current location can be used as location source. Either a cellphone with GPS, rfid which is tracked at specific gates, or using the GSM

network. For packages that are mostly stationary rfid and update requests that are only sent when a package is actually moving. But cars or other objects that are not transported within something else should have their own should not depend on other devices to send their updates for them.

4.10 Home zone selection

New users are required to select which zone should be designated their home zone. To select a home zone the user will have a list of the servers and their geographical location. Then they can calculate the server closest to their current location, and chose that as their home zone.

4.11 Network protocol

To communicate either between nodes in the network, or client and servers TCP will be used. TCP is statefull which, but it does guarantee that the datastream arrives in the correct order.

The datastream is a series of message. Messages are test strings ended by a newline, and they are completely self contained. This allows a connection to remain stateless. So if a connection is dropped, there are no extra steps required when reconnecting.

The reason for keeping the communication in pure text is so there is no need to convert between byte formats which may differ between different hardware. This systems has to support a range of different devices and byte representations will be different on them.

4.12 Server discovery

The clients must always know which servers belong to which zone, how many zones are available and which area they cover. Because the clients are responsible for connecting against the closest zone and staying connected to a working server within that zone.

Clients will be distributed with an initial server list, which can be used for the initial connection to a zone. Since the zones are already interconnected they know what servers are online and which zones are present. Which allows the user to request an updated zones and server list when it first connects.

4.13 Server overload

A server could get too much data to process if too many clients try to send requests at the same time. The clients should detect that the server uses more time than usual to respond and therefor possible connect to a different server within the zone for the next requests.

The server is also allowed to disconnect clients when too many are connected to make sure there are not too many connections or too many requests being sent.

4.14 Use cases

Here we will describe a few common scenarios. How they will propagate information through the system, and how the zones will interact with the clients.

4.14.1 New user

The first action a new user will do is locate his home zone, so he can generate a user on the correct server. The user id that it receives will be stored on the client so it can identify itself later on. The new user is during registration stored on the server. Initially storing the user without any location information is done to allow for other users to subscribe to the user's updates even before the for location information is pushed from the client.

After registration the user pushes its first location information to the server, which is stored in the `users` table described in section 4.4.

4.14.2 Subscribe to updates from a single user

To send a subscribe request a user first connects to the home zone of the user he wants to subscribe to. The request is then sent to that server, which adds a new entry in the subscriber table.

If the user that is be subscribed to is not currently located in his home zone a new request is sent from the home zone server to the zone the user is located in with the new subscriber information.

4.14.3 User update with only local subscribers

A user with subscribers sends a location update to his current zone. The location is stored in the server database and distributed to the other backup server on the same zone.

The list of subscribers is collected and all the subscribers also have current zone as their active zone. The list of subscribers is then matched up against which users are polling for updates on the zone. Each of those subscribers is sent the new location.

Subscribers may also be located at another zone than their home zone, and the table which contains the current zone for each user must also be consulted. A list of users not located in the home zone will be created and grouped by the zone they are located at. And a request is sent to each of the represented servers with the update information.

4.14.4 User update with foreign subscribers

The location update is sent from the user and handled in the same way as if there were only local users. Except that the subscriber list is grouped by which zone each subscriber belongs to.

When the subscriber list is retrieved and grouped the server sends a request to each of the server represented by the subscribers. They will receive a list of all the users which are subscribing for the update, and the new location information.

The list received by each server is then compared to the users that are waiting for updates and a request is sent to each of them with the new location and which user is belongs to.

4.14.5 User update in foreign zone

A user that is not in his home zone when sending a update request will be automatically migrated to the zone he is in if that has not already been done. The steps taken for the migration are shown in subsection 4.14.8.

If the migration is not disallowed the location is stored on the current zone, and the information will be sent out to the subscribers as described in subsection 4.14.3 and subsection 4.14.4, except that the `visiting subscribers` table is used instead of the `subscribers` table.

On the other hand of the migration request is denied the information is passed on to the server currently responsible for the user that sent the update request. And that server will then restart the process of storing and sending out the information to any possible subscribers.

4.14.6 Home zone server downtime

Either during interaction, or when connecting to the home zone the server might fail or be unavailable. If that happens the user retries the request against one of the hot spare servers in the server list for that zone. The users should stop sending a request if updated location information is available. Instead a new request should be constructed with the new information.

4.14.7 Waiting for user updates

For a user to receive the latest location updates from the users he subscribes to, he must connect to his closest zone and send a poll request. This will trigger a migration request if the server he connected to is not currently responsible for him. If this request is denied he must connect to the zone that is responsible for him until the migration can be done successfully.

When the user is connected to his responsible zone and the poll request was successful he will automatically receive all new location information that he subscribes to.

4.14.8 Migrating from a home zone

As a user goes from their home zone to a neighboring zone the data belonging to the user will be duplicated, the zone the users has moved into will contain the master copy of the user's location after the first update request is sent to the new zone the user arrived in.

A migration can either be started by a update request, or when a user polls for updated locations from the users he subscribes to.

When a migration is started the new zone contacts the users home zone to take responsibility for the user, this results in the home zone sending over all the users that subscribe to the migrating user and the user's current location. The home zone will locally mark which zone the user now is located within, and the current time of the migration.

4.14.9 Migrating to a home zone

Migrating back to a user's home zone is less work than going from the home zone to another. The home zone already has the latest subscriber information. The only information that must be sent from the current zone to the home zone is the user's location. After that is done the current zone can allow the home zone to take over and delete all subscriber and location information about the migrating user.

4.14.10 Back and forth migration

A user migrates from one zone to another, and suddenly turns around and goes back to the previous zone. This actions triggers two migrations right after each other. After the first zone crossing a migration is triggered, but because the second migration is initiated too soon after the previous migration it is denied.

The user then has to reconnect with the previous zone and keep sending requests to it until a migration is allowed to the new zone.

4.14.11 Entire zone downtime

An entire zone goes down, and all the connected users are therefor disconnected. A few of these users are only visiting the zone, the rest of the connected users has the zone as their home zone.

The visiting users search through the server list for the closest operational zone and connect to that one instead. This sets of a migration for the visiting users which will be able to continue working correctly.

All the users which had the zone that went down as their home zone must wait until the zone comes back up. If they were to connect to a new zone their migration would fail because the new zone would not be able to fetch the required information from their home zone.

Any user with the failed zone as their home zone that were located away from it will be able to function properly within the zone they are currently connected to. Trying to migrate will fail for the same reason as above.

Chapter 5

Discussion

5.1 Multiple zones

Having multiple zones where each is responsible for their geographical area was decided on for several reasons. It keeps communication local and avoids sending data over long distances. Close geographical location also keep network failure problems to a minimum, even if a zone is not reachable by the rest of the world clients within that zone can continue to operate as if nothing happened.

Splitting the world in several zones requires separate server centers for each zone, which results in maintenance overhead. But because this system requires high uptime, having redundant servers in different location is crucial, and the maintenance overhead is there anyway.

5.1.1 Zone failure

When a zone fails it is not possible to communicate it from the other zones. This will result in some users not being able to migrate and some need to change zone before they can continue to send update requests.

A temporary zone failure is unlikely when there are redundant servers with fail over capabilities. And in case of massive network failure within that zone the devices would not be able to communicate with the rest of the system anyway. The major problem with zone downtime is that it denies users with the failing zone as its home zone to migrate between two other zones. But in case of temporary failures continue to us a zone you have exited is not a huge problem for a short amount of time.

The other problem is that the zone will no longer be able to relay update requests to polling users outside the zone. But the users which is most interested in the updates is probably within the same zone. And either update requests from the users does not reach the servers because of server failure, and there is nothing to relay. Or it is just the external network connection that is the problem and the users within the zone is able to continue to work as if there were no problem.

5.2 Mapping location to zone

There are multiple ways to chose which zone to connect to depending on location and connectivity. Choosing to have a list of zones, servers and geographical locations to decide what zone and server to connect to was decided as the best solution when looking at simplicity, connection quality and physical distance to the servers.

The clients could auto detect which server to connect to depending on latency between the client and server, or number of jumps between them. But this adds quite complexity to which server should be selected, as well as failing when there are temporary network failures which routes packets around the problematic router which could result in high latency for short time periods. Specially when select a users home zone this could be bad since the home zone selection can not be changed at a later time.

Anycast is a way for IP packets to be routed to the closest server which belongs to that IP address. It could be used to automatically choosing close servers, but the client would not be aware of which servers it connects to, and if a server goes down it would instead connect to a server in a different zone than intended and the client would therefor not be able to fallback to a reserve server in the correct zone.

Sending a server list with each client allows first for all for bootstrapping the connection to the location database, and it can be periodically updated to always have the latest server information.

A server list also makes sure the client is using the zone it is closest to geographically, which will probably also be the zone which it will have the best connectivity against.

5.3 Database layout

The database tables were kept separate to hold the least amount of information in each of them. The current zone could be a part of the location information for a user, but retrieving that information is not necessary in most cases, and the update frequency of the two pieces of information is very different. Location is updated often, while the current zone is rarely updated.

Splitting up the data in visitors and users with that home zone is not necessary, but is done because of the different characteristics of the data. For the users with the zone as their home zone the data must be properly stored on permanent storage. The subscriber list for visiting users on the other hand does not require any permanent storage since it can be retrieved from the visiting user's home zone.

Storing a list of subscribers is chosen instead of storing a list of the users a subscriber subscribes to. If a list of user someone subscribes to had been stored at each subscribers home zone a traversal of the entire user base would need to be performed for each update. With a list of subscribers connected with the user they subscribe to all the information is available on the zone the user sends his update to, and which users should receive the update is easily collected.

Storing the current zone of a user in his home zone is an optimization done

because the user needs to be located both when someone sends in an location update the user has subscribed to, or if a new subscriber is added to that user. The user could in these cases be looked up by doing a search across the other zones, but it is faster if the home zone always knows where each of its users are located.

5.4 User id

A unique identificator for each user is needed to tell users apart across the network. Because of the decentralized nature of the system a part of the user id must be created on a zone without talking to the other zones in the system. Both because of the global lock that would incur on the user creation process and that two zones might temporarily not be able to talk to each other.

By splitting the user id up and having two parts of it, where one represents the home zone of the user, and the other is a unique id for a user within that zone. The user id is guaranteed to be both unique across the entire system, and no communication across zones are required.

Having a user supplied identificator is the norm on services today. And it gives a personal feel for user, but is not with any practical importance. Users are already connected by email or other communication media, and sending a user id with two numbers is no different than receiving a handle or nickname which denotes the same thing. Having an alias is therefor not supported in the system described.

To avoid sending back and forth the user supplied identificator between the servers and zones, the identificator could be resolved internally to a unique number which then could be used efficiently.

A third option for identificators is to have a user supplied identificator for the zone unique part, and still prefix it with the zone identificator. This would allow the user to have some relation to his user id, and still not require cross zone communication when creating a new user. But to avoid the extra check and avoid variable length for user ids the encoding with two numbers is the best for the in this database.

5.5 Home zone

The home zone is the centralized part of the system, and every user has a zones he belongs to. This makes each user dependent on one zone, which can be problematic if the zone is unavailable. Another possibility to keep the system entirely decentralized is to always have the current zone for the user be responsible for him.

The problem with not having one home zone for each user is that a discovery mechanism must be used to find each user. Using a distributed hash table distributed across all the servers would allow for mapping from user to zone, but a server that has nothing to do with a user might end up holding information about his location.

Using a home zone the migration protocol is not as complicated as it will be without a master. With one master which is responsible for all migrations for that

one user, the zone has all the current information for the user and it will always know where he is located and between which two zones he wants to migrate.

Having a home zone simplifies the entire design of the system by having a master for each user that exist, and adds locality to the communications.

5.6 Subscription

The list of users that will receive updates from the users that update their location is both stored at the updating user's home zone and the current zone for the user that sent the location update. The home zone holds the authoritative subscriber list to make sure the data is stored on multiple servers and allow updates when a user is not in his home zone.

Instead of always going to the home zone of a user to subscribe to him, new subscribers could go to the home zone only to find out which zone a user is currently in. And then go there to subscribe to location updates. The problem with that solution is that a subscriber potentially must contact several servers to subscribe to updates. Because only a home zone holds redundant copies of a users data subscribers could also be lost if a server died before it could replicate the updates over to a user's home zone.

When a home zone is responsible for keeping the current zones of users updated with the latest subscriber lists the data will be stored and durable when a subscriber receives a successful reply after subscribing, and a server death will only delay the updates lists to reach the current zones.

First storing new subscribers on the home zone and then sending out updates to the current zones adds a small delay even if everything works. This could lead to some updates that are sent after the subscriber subscribes to a user not reaching the subscriber. This delay is acceptable to ensure that all new subscribers are stored properly and will not vanish if a server dies.

5.7 Network protocol

There are several network protocols that work on top of IP that could be used to communicate between entities on a network. They all have their strong and weak points, but TCP ended up being the best fit for this system.

5.7.1 Transmission Control Protocol

Transmission Control Protocol[9] is a connection oriented communication protocol. In order to communicate using it the two endpoints must for create a connection between each other. TCP depends on the client-server model where one part waits for a connection request, while the other initiates the connection. The protocol is also responsible for making sure each packet arrive at the other end, and that the network packets are assembled in the correct order to make sure the data stream is the same on both sides.

The connection oriented design of TCP adds extra data overhead, and connections will end up stalling if there is a lossy link since the lost packets must be resent and sent to the receiving application before the later packets can be used.

5.7.2 User Datagram Protocol

User Datagram Protocol[10] does not support permanent connections, but packets are still sent to and from different ports which can be used to separate different endpoints. Each packet is independent of the other packets sent to and from the same pair of ports. This means that one packet might appear on the endpoint before one sent prior the the arriving one, but packets are not guaranteed to appear at all, which can be problematic if you need all data to received.

The connectionless design of UDP allows for smaller overhead when sending packets, but to guaranty that the required packets do arrive an acknowledge protocol must be implemented, which complicates the application could, but could result in a more efficient protocol since it does not have to support all the corner cases general purpose connection oriented protocols support.

5.7.3 Stream Control Transmission Protocol

Stream Control Transmission Protocol[11] is as TCP connection oriented. But it supports multiple streams within one connection, where each stream does not interfere with the others. A packet sent in stream X does not have to arrive before a packet sent in stream Y even if it was sent prior the packet in stream Y.

Data sent from an application to SCTP is handled as messages, two different messages will never be represented in the the same SCTP packet. This allows the receiving application to get messages out of order, but always get the entire message assembled in the correct order.

5.7.4 Comparison

The communication protocol used between the servers and clients are message based, which would work very well with the way SCTP is design, SCTP also perform better than TCP in some cases[12]. But SCTP is not properly supported on all devices yet which TCP and UDP are. Nor have the implementations been tested in the same way as the two older protocols.

UDP has little overhead, but requires the application to handle much reassembly of packets which has been split up, and reorder packets which does not arrive in the correct order.

For the database communication TCP is the used protocol. It reassembles byte streams in the correct order for the application, and makes sure all the packets arrive at the end point. TCP is also a thoroughly tested on all platforms connected to the internet.

5.8 Full mesh communication

Having a full mesh where each zone potentially connects to every other zone may result in many connects when the number of zones grow. But since zone are spit up based on geography most friends of a user will be located within the same zone as the user they subscribe to.

To avoid connecting multiple times between two zones each zone could designate one server which should communicate with one zone and relay all information to that zone through the designated server. Relaying can also help across zones. So a group of zones decide that one of the zones should be responsible for communicating with another zone farther away.

5.9 Server overload

Handling and overloaded server can be done either by a way to communicate with the users that they need to send updates with greater pause between each update, or add a artificial delay before acknowledging their update and by doing that delaying their next update.

Another way is to allow the client itself to determine when a server is overloaded. This solution was used because it can be tied in with the detection the client must do to determine if a server is currently unavailable.

Letting the server disconnect clients as well is an extra step taken to make sure a server is not getting too much requests before the clients notice that the server is actually overloaded.

Chapter 6

Implementation

A server has been implemented using C to test communication between server and client, and to implement a communication protocol. To avoid implementing a backend from scratch, a embeddable key value database was used called BerkeleyDB(BDB)[13]. BDB supports storing arbitrary binary data which allows us to send it structures directly from C without encoding it in any way.

The server was implemented as a threaded application where there is a thread running which accepts connections and network requests. These requests are then parsed and added to a request queue. This request queue has several consumers which takes the oldest request and handles it.

6.1 Storage

Using BDB for disk storage removes the burden of storing the data efficiently and gives a simple API for storing and retrieving the data. BDB makes it possible to add new indexes to the database if it is required to look up locations based on other information than the user id.

The user location is a structure which is stored sent directly to BDB. This is problematic if the database must be moved between servers or if the architecture is changed. Because floating point numbers is not always represented in the same way, or if a computer architecture requires another alignment of the data the data would not be the same any more. Separate exporters can easily be written if a database must be transferred, and having a binary format avoids the overhead of converting the data.

6.2 Protocol

A text protocol is used for communication between network programs. A binary protocol was avoided to avoid problems with different encodings of numbers and endianess. And text is easy to debug and find errors in the request sent.

```
Client: update user "1-5" [-10.1035 9.1249]
Server: #[13.1925 -15.4324]
        SUCCESS
```

Figure 6.1: Update user command example

```
Client: fetch "1-3"
Server: #[13.1925 -15.4324]
        SUCCESS
```

Figure 6.2: Fetch command example

6.2.1 Commands

All commands sent will return a reply. If the command succeeds the string `SUCCESS` will be sent back, in case of a failure `FAILED` will be returned.

To make parsing easier keywords are separated from input strings, and location has it's own format. Keywords are ordinary words, like: `keyword`, they can be both upper and lower case. Input strings are strings that are not know during compilation of the program like user names. They are written in quotes(`"user id"`) to allow the parser to separate them from keywords. Locations are written inside of square brackets which contains two floating point numbers(`[`5.1024 -10.3532]).

Update user

Updating a user's location is done by sending `update user` to the server. Two arguments are required, the first is the user id of the user that has a new location, and the second is the new location. An example is given in Figure 6.1.

Fetch

The `fetch` command returns the latest known location for the user given as an argument. An example is given in Figure 6.2.

Subscribe

Sending `subscribe` to the server will subscribe the user id given as the first argument to all updates from the user id given as the second argument. An example is given in Figure 6.3.

```
Client: subscribe "1-4" "1-5"
Server: SUCCESS
```

Figure 6.3: Subscribe command example

```
Client: subscribers "1-4"
Server: "1-2"
        "1-6"
        "1-8"
SUCCESS
```

Figure 6.4: Subscribers command example

```
Client: poll "1-5"
Server: SUCCESS
[time passes]
Server: newlocation "1-5" [56.5116 -10.1503]
```

Figure 6.5: Poll command example

Subscribers

The `subscribers` command returns all the subscribers for the user given as an argument. Each subscriber is returned on its own line. Example given in Figure 6.4

Poll

Sending `poll` to the server tells the server that you are logged on and awaiting updated location from the users you are subscribed to. Poll takes one argument which is your user id. A usage example is given in Figure 6.5.

New user

The `newuser` command creates a new user and sends back the user id for the new user. An example is given in Figure 6.6.

6.2.2 Parsing

The protocol is parsed using Bison[14] and Flex[15]. These are a set of efficient tokenizer and parser generator which allows the protocol parsing to be easily extended in the future. They can also generate reentrant code which allows the server to parse multiple requests in different threads at the same time.

```
Client: newuser
Server: "1-10"
        SUCCESS
```

Figure 6.6: New user command example

6.3 Request queue

The request queue is represented as a linked list where new requests are added to the end of the list, and the consumers consume it from the head. The queue is designed to take any type of work and hand it over to a worker thread. Each added element to the queue has a function and some data. The function is called by the worker thread with the data as the function argument. This allows the program to spread the work load across several CPUs and cores.

Having a queue with waiting requests and actions helps to figure out the load on that server. Keeping an eye on the queue the server can know if some clients should be redirected to another server or if the server can take on even more requests.

If the request queue is empty the consumer threads will use the threading library to wait for new requests to be added. This blocks the consumer threads until there are new data to handle so it does not spin and use CPU without doing anything useful. When new data is added the adder will issue a signal to one or more threads telling them that there is new data available.

6.4 Threading model

The threading model chosen for the server software to let the software scale up and support several CPUs and cores. Every action that requires work is queued up and handled by one of the worker queues.

This does adds some complexity to the code. Everything must be thread safe, and all the sockets that are used must be properly locked before they are read from or sent to. If they are not data might end up mixed with each other.

There are also other ways to take advantage of several cores. The program could make a new copy of itself for each new connection. But this will end up using more memory, and communication between the different copies of the program requires more complexity. Or a one thread per connection could be used, but that will spawn more threads than necessary since the clients spend a lot of time idle, and the context switching between threads would add an extra cost.

Because of the heavy use of threads and concurrency in the server there is quite a bit of locking done to deny concurrent access to the same object. Specially the request queue has a lock around itself which only allow one thread to work on it at the same time. This denies a consumer access to one side of the queue while the network thread is pushing a new request on the queue. Which can result in heavy lock contention. But the work done on the queue is much less than the work done by the threads before and after they work on the queue. And this should balance the queue access out with a few threads, but if there end up being many threads some sort of lock free queue should be tested to see if that ends up being more efficient.

6.5 Network communication

When a new connection is established the server stores the new client in a hash table and adds the socket to the list of sockets which should be checked for activity. The client information contains a lock which is used to serialize access to that client socket. When the client lock is held the server can send data to the client, but all received data must pass through the network thread with is responsible to reading.

Chapter 7

Future work

7.1 Find nearby friends

A huge improvement of the system would be to locate users located near your current position. By having a simple way to locate the people nearest to your current position you could figure out which friends are nearby, or finding new people with similar interests as your own to meet.

7.2 Authorization and authenticating

For the system to usable by a large audience some security measurements must be put in. A user must be able to authenticate, and updates to that user must only be allowed if the authentication succeeds.

A user should also be able to allow or deny subscriber requests. Either before any updates are sent to the subscriber, to allow for complete control over who gets to see his updates. Or after subscribers has been added to deny specific people access to his updates.

Encryption for the communication channels is also an important aspect to look at if the data sent if somehow confidential, or the users them self want to limit the number of people that can read their location even more.

7.3 Outlier detection and correction

Devices that finds your current location is not always reliable, measurements might be off a little, or very much. This should be detected and warned about, or update requests that are obviously wrong could be denied.

This could be achieved by finding the velocity of the user, and if it is above some threshold the update could be denied. Or by using a Kalman filter[16] to find more probable values for a given update based on the other updates from the same user.

7.4 Keep historic data

A huge improvement to the described system would be to include historic data, and give users the ability to traverse the history and get positions from a given time frame. This would allow for plotting of user movement from last year, or last week. As well as going back and figuring out where someone was at a given time.

Chapter 8

Conclusion

A system allowing users to continuously send in their current location which can be retrieved by friends or family. Other users has the possibility to subscribe for updates, and when they are connected to the network they will receive updates in real time when the users they are subscribed to updates their location.

The described system is able to scale up and handle massive amounts of concurrent users. Using a geographically disperse setup allows the system to continue to work even when there are connection problems or an accident which take down one or more servers.

A linear scaling for the system is accomplished by either splitting up one overloaded zone into two or more, or by adding more servers to a zone. This allows for continuous operation without having to invest in more hardware than there are new users.

Independent zones are important to keep the system available. The only communication between zones are when they are on behalf of the users. By keeping the cross zone communication to a minimum a zone can be unavailable for the rest of the system without it degrading the system as a whole. The only affected parties are the users which is not able to migrate or receive all the location updates.

Latency has been used as the primary constraint for how this distributed system should be designed. This gave the advantage of both closer distance between the communication end points and the system avoid relying on a global working network connection. To avoid too much data being sent for the client all information sent that should be distributed to several zones are relayed by the zones them self, which ends up cutting the bandwidth needs for the client.

Looking at the different possibilities for how the user should select which server to connect to it is shown that having a set of server for each geographical location is the best solution when working with a location system. Having a home zone which is responsible for the user ends up being the best way to have control over the user movements and have an authoritative copy of all the user data.

Bibliography

[1] G. DeCandia *et al.*, Dynamo: amazon's highly available key-value store, in *IN PROC. SOSP*, pp. 205–220, 2007.

[2] D. Karger *et al.*, Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web, in *In Proc. 29th ACM Symposium on Theory of Computing (STOC*, pp. 654–663, 1997.

[3] Facebook user statistics, `http://www.facebook.com/press/info.php?statistics`, 2010.

[4] M. Paksula, Persisting objects in redis key-value database.

[5] J. Scourias, Overview of the global system for mobile communications, 1995.

[6] M. Mouly, *The Gsm System for Mobile Communications* (Pearson/Prentice Hall, Upper Saddle River, 1992).

[7] mbuddy logg inn, `http://www.mbuddy.no`, 2010.

[8] D. E. Ericksen, Nad83: What is it and why you should care, 1994.

[9] J. Postel, Transmission Control Protocol, RFC 793 (Standard), 1981, Updated by RFCs 1122, 3168.

[10] J. Postel, User Datagram Protocol, RFC 768 (Standard), 1980.

[11] R. Stewart *et al.*, Stream Control Transmission Protocol, RFC 2960 (Proposed Standard), 2000, Obsoleted by RFC 4960, updated by RFC 3309.

[12] N. G. Rajesh Rajamani, Sumit Kumar, Sctp versus tcp: Comparing the performance of transport protocols for web traffic, 2002.

[13] Berkeleydb, `http://www.oracle.com/database/berkeley-db/index.html`.

[14] Bison - gnu parser generator, `http://www.gnu.org/software/bison/`.

[15] flex: The fast lexical analyzer, `http://flex.sourceforge.net/`.

[16] G. Welch and G. Bishop, An introduction to the kalman filter, 1995.