PROJECT SUMMARY

Concept map is a method to represent knowledge in a graphic way. One of its usages is that people can share their ideas over by editing the same concept map. However, current concept mapping tools usually don't support remote edit of concept maps, the few ones support it are all centralized in design. This makes it difficult and inconvenient for distributed concept mapping, which means distribute the editing of the same concept map over the whole network to allow remote and concurrent concept mapping.

In this project, a decentralized distributed extension is implemented for distributed concept mapping. It doesn't require the user to share their concept map manually or relying on a central control system.

The unique combination of the extension design brings flexibility and convenience, but it also brings challenges. I argue that the network the extension forms is unstable in structure, thus, existing methods on a distributed network will fail to improve the performance. Research is done on these methods and a stability parameter is introduced to improve commitment ordering and load-balancing in the network, and also the stability makes them fit for the unstable environment. In addition to the improved methods, a modified version control method is also used to support the improved methods.

- According to the research done in this project, I added stability parameter as a measure to fit the existing methods to the extension environment. See pages 13 for definition of the stability.
- I modified the commitment ordering technique with the stability parameter and implemented it in the extension. See pages 14-16 for details.
- I modified the load-balancing technique with the stability parameter and implemented it in the extension. See pages 16-18 for details.
- I modified the version control technique and implemented it in the extension. See pages 18-20 for details.
- I implemented an extension for distributed concept mapping, each of this extension hosts several users and allows them to work on the concept maps remotely and concurrently with other users on the same or different extensions. See pages 41-47 for detailed summary of functionality.
- Testing and evaluating of this extension shows it is adaptable to the changes of the network. With a reasonable performance in a relatively unstable network.

Acknowledgement

I would like to thank my supervisor Dr. Steve Gregory, for guiding me all along my project and my dissertation.

Thanks also to my family for their support when I was working on my project.

INDEX

1 INTRODUCTION

1.1 AIM OF THE PROJECT

The aim of this project is to supplement Conception with a distributed extension which allows distributed and simultaneous edit of concept maps with reasonable operation efficiency.

1.2 OBJECTIVES

The main objective of this project is to implement a distributed extension for Conception. However, the extension is designed to form both distributed systems and peer-to-peer network. Such design brings about problems such as concurrency control, data synchronization, and load-balancing. Also, the network the extensions formed will tends to be unstable since each user joins and exits it by their own will. This combination makes this project challenging and existing methods are required to be improved for this. The improved methods of commitment ordering, load-balancing, and version control which are essential to the extension, are stated in chapter 3.

The implementation of the distributed extension is a challenging software engineering task itself. The first step will be to design the implementation based on the research results of the first part of this project. The detailed design of this extension is given in chapter 3. The final design of the extension retains both performance and flexibility. It can be used in two different modes for different circumstances. Though the external structure of the network formed can be centralized in some part, the internal network structure is totally decentralized.

The final part of the project is to test the extension's functionality and evaluate the improved methods performance. The results are shown in chapter 4.

The objectives of this project are:

- Improve the commitment ordering model to fit unstable network environment
- Apply load-balancing mechanism to the network formed
- Design the extension with the improved commitment ordering and load-balancing
- Implement the distributed extension

- Evaluate the extension and the methods

1.3 BACKGROUND

1.3.1 CONCEPT MAP

According to the developer of concept maps, Joseph D. Novak, a relatively formal definition was given in [1], "Concept maps are graphical tools for organizing and representing knowledge."

Generally, a concept map is a diagram containing the many concepts contained in the abstract knowledge, and the relations between them. It is a very useful tool for representing knowledge in a graphical way.

The figure below is a very simple concept map on part of this project.

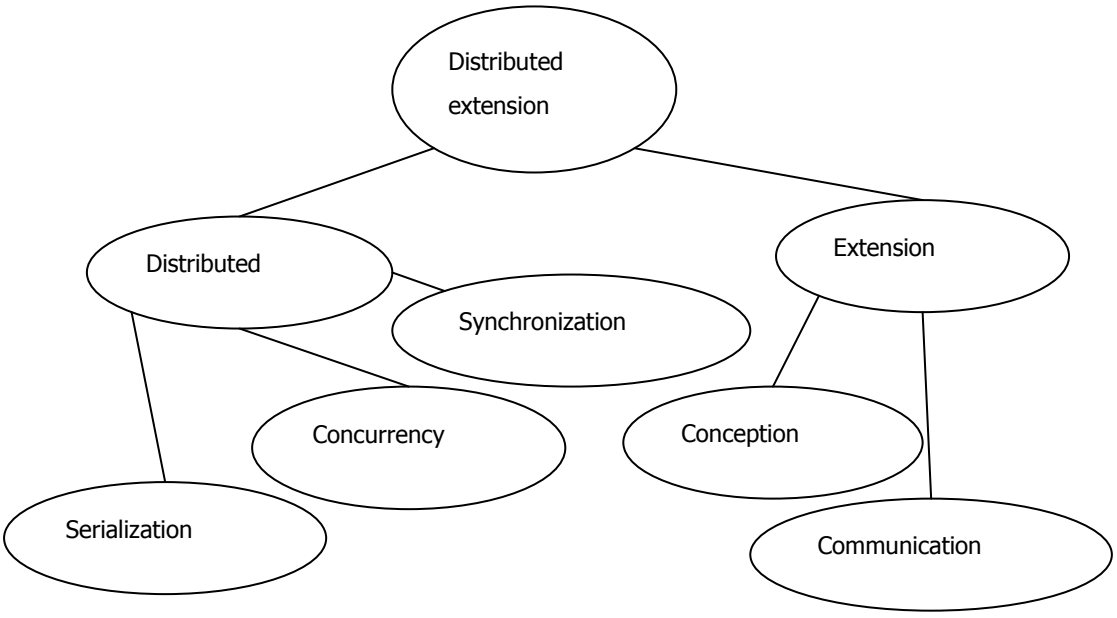

Figure 1-1 Simple example of a concept map

According to the research of Novak, Concept maps can be a powerfully tool for both education and evaluation. [1]

With an extension to allow distributed editing of concept maps, the usefulness of concept mapping tools can be improved considerably. All the operations on it will be

much more convenient and flexible, and people will always benefit from the convenient and flexibility.

1.3.2 CONCEPTION

Conception is the software to be extended in this project. It is a concept mapping program developed by Parallel Logic Programming.

The use of Conception is usually limited to a local or internal network such as a company network where the identity of each client is clear and the data transferred are supposed to be confidential, so this project will focus on solving the concurrency load-balancing problems with less concern on security factors.

1.3.3 DISTRIBUTED SYSTEM

According to the definition in [4], a distributed system should contain several independent computers which cooperate with each other on the same work, and the users in this system should have the feeling that they are using a single system.

The distributed network is the main component of a distributed system. The distributed network in this project is constructed mainly for distributed editing of concept maps. It controls the commitment of the concept map versions, which are the main component of a concept map in the extension. Whenever a commitment is needed, either automatically or by user issued command, the distributed system detects the conflicts and prevents most of them by commitment ordering. This happens prior to actual commitment request, and the actual request is blocked until it is its turn or it has reached timeout threshold.

Each concept map is edited by several processes in different computers. Thus, each of these computers is considered a node.

Each concept map is edited by several of these nodes, and they form a distributed system in which the resource is managed within the corresponding concept map. Several of such distributed systems co-exist over the whole network. However, only one actual distributed system exists in the extension network. Thus, the distributed systems mentioned above are all considered sub-systems of the actual distributed system.

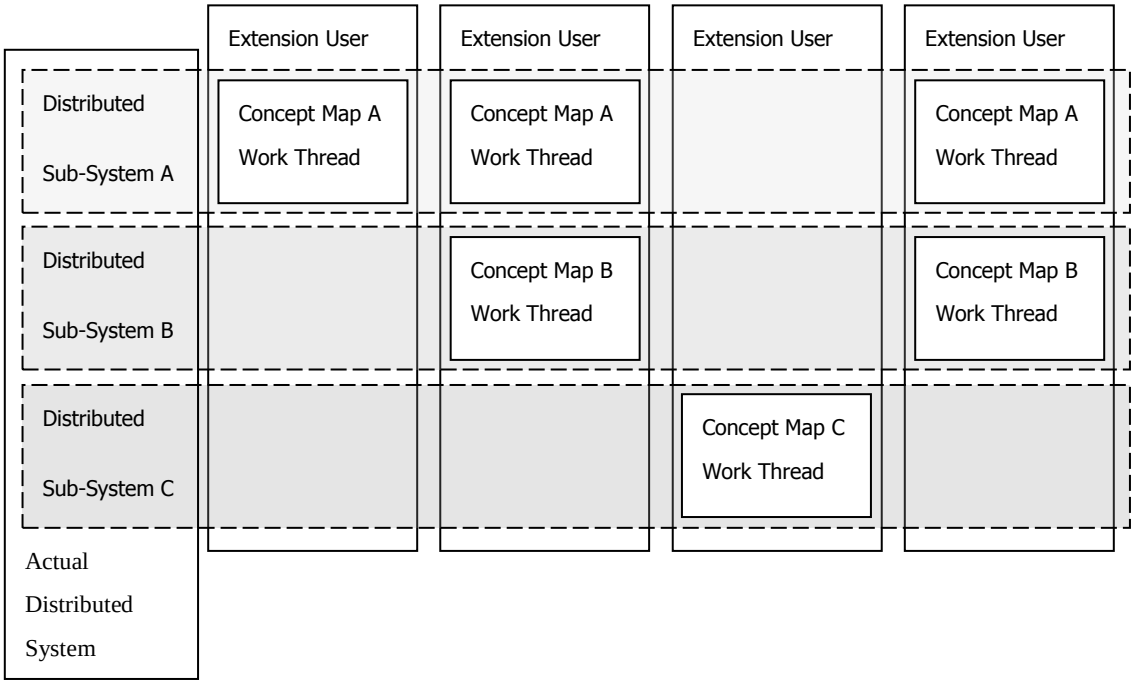

Figure 1-2 Distributed system formed by the extension

As shown in the above figure, the edit of the same concept map is distributed among several users. However, user joins and exits one of the distributed systems by their own decisions since it only depends on the software running on their computers.

The extension is designed to distinguish the communications of different sub-systems. Also, each of these sub-systems has its own set of resource and control, all co-exist within the only actual distributed system. This is a flexible design, so that a user can open and edit several concept maps at the same time.

Users may try to commit their edit just like they save the changes made, the commitment ordering happens in one of the distributed sub-systems, and is actually handled by the only one control system over them.

1.3.4 COMMITMENT ORDERING

Commitment ordering is a concurrency control technique. It was originally introduced by Yoav Raz in [5], as a technique for database concurrency control. The definition is given in [5], "Commitment Ordering (CO) is a serializability concept, that allows global serializability to be effectively achieved across multiple autonomous Resource Managers".

Yoav Raz also stated in [6], commitment ordering "allows to effectively achieve global serializability across multiple autonomous Database Systems, that may use different (any) concurrency control mechanisms".

Essentially, commitment ordering serializes all commitments, determines the order of the sequence of commitments to be made. It helps maintain the correctness of commitment results by preventing the conflicts between them. Though originally proposed for very large databases, this technique can also be used in any distributed systems where the management of resource is needed, such as this extension.

"CO generalizes the popular Strong-Strict Two Phase Locking concept…. While S-S2PL is subject to deadlocks, CO exhibits deadlock-free executions when implemented as nonblocking (optimistic) concurrency control mechanisms".

The concurrency control happens to be implemented as a partially non-blocking one in this extension. Thus, commitment ordering will better fit for the extension than the SS2PL model as it can avoid deadlocks in the non-blocking parts.

1.3.5 VERSION CONTROL

Version control, also known as revision control, is a resource edit management method mainly used in software engineering. It utilizes synchronization mechanisms to ensure the documents are correctly modified. The idea of version control is to allow several concurrent edit process being taken on the same file without actually change the data until it can be confirmed.

Whenever version control is needed, a branch version is created for edit. At the end of edit, it can be confirmed and merged to the main version. Then the main version is updated. Each user edit on the base version generates a new branch version, the newer branch versions are then being committed and the main version updated.

Version control ensures every change on the resource is executed and only the newest version is stored. It naturally is fit for distributed environment because it allows concurrency.

1.3.6 PEER-TO-PEER NETWORK

All users of this distributed extension network are treated equally in that they are all using the same extension which gives the same privileges and functionality regardless

of its configuration. Thus, to be more precise, the distributed network in this project has a peer-to-peer network layered on it.

Initially, the network is not necessarily considered a peer-to-peer network when only the version commitments exist. However, since the distributed storage and transfer of data also resides in this network as an essential part, the properties of peer-to-peer networks must also be taken into consideration.

Since each of the clients holds copies of one or more concept maps, the stored concept map can be seen as the resource shared by the extension. Like any efficient and common peer-to-peer file sharing software, the extension uses distributed hash table for the indexing and searching of the concept maps.

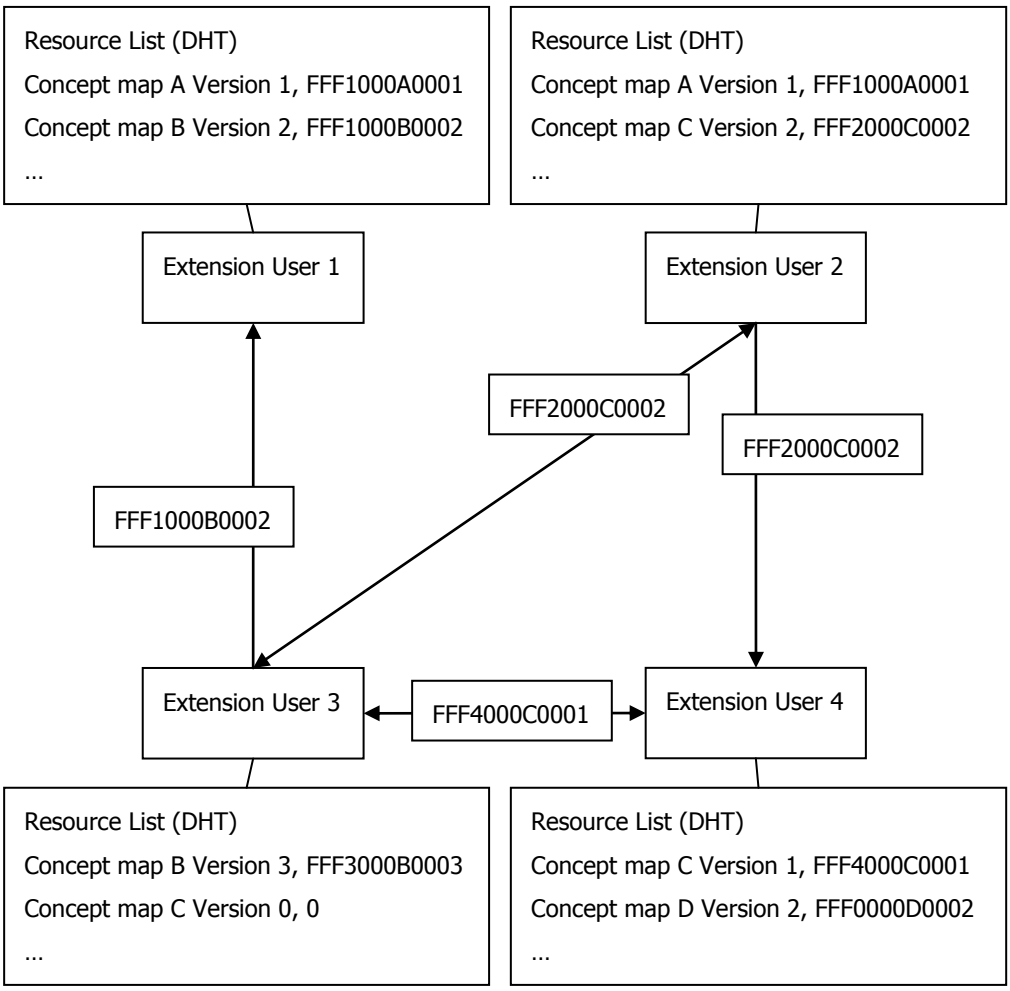

Figure 1-3 Peer-to-peer network formed by the extensions

Extension users are the nodes of this peer-to-peer network as same as in the distributed network, and different versions of concept maps are resource shared between nodes. The figure below shows the process of resource searching and transfer. Each extension have its own resource list, which is a distributed hash table, it records where part of the resource in the network resides (Actually, it mostly records whether the data has a local copy; the exact location of other nodes' resource needs further confirmation). The information used for query and transfer is the hash code of the actual resource.

The peer-to-peer nature of the extension network brings several benefits:

- All elements of the network are equal means it is better to be totally decentralized. Thus, the scale and capacity of the whole network only depends on the number of users. As more users join the network, all resource in the whole network increases.
- Decentralization also increases the average data reliability of the whole network, since data is stored with high redundancy.
- Each extension acts as both a server and a client; users can be given much more control over their resource.

However, it also has its drawbacks. Some of them are listed below:

- The benefit of increase in average data reliability only applies to the data reliability and actually caused instability to the structure of the whole network as the nodes of the network acts freely.
- The network traffic in peer-to-peer network are all self-motivated and without the knowledge of global traffic distribution. They may rush to the few more preferable nodes for resource while other less-optimal nodes also hold it.
- The nodes are given more freedom, which will result in more problems when it comes to management concerning global status. More complex algorithms are needed for these controls.

1.3.7 LOAD BALANCING

The main goal of load-balancing is to balance the traffic distribution in the whole network. However, in the decentralized environment, each node only has limited knowledge of the whole network. Thus, it is natural to consider it from both the request and receive users' perspectives:

Normally, when a request needs to be sent, the request user doesn't know whether the target is busy or not so he choose to not care about it. The request is then sent to the first available node at the request user's ease.

The receive user gets this request and do what it requests. However, one of the receive users may get many requests over a short time because most users prefer him as the target. Thus, he needs to work with much extra effort to reply all of them. Then this user will want to prevent itself from being too busy with the received requests, he may choose to restrict itself to only work within his work limit at one time. He may then work at an acceptable pace as he wants.

However, the user sent the excluded requests are still waiting for this user to reply. If left alone, their efforts of waiting until their patient limit will be all in vain. This is unacceptable to the responsible receive user. Thus, he tries to get all the work done while not conflicting his work limit.

He tries to schedule to handle the excluded requests to the next time slot available. But at this time, requests are coming all the time. There are always more requests coming, the schedule grows so large that most of the request users reach their patient limit before the reply is sent. It more wasteful than leaving them alone because the receive user put efforts replying them.

Since all the users are equally treated, a reply from any user who can reply the request is the same as one he replies. Thus, he tries to send the excluded requests to the first node he finds available. Like any other request nodes, he doesn't know and doesn't care the target node's condition. This way solves the problem but actually more than one request is made for one request to be replied.

The paragraphs above provides three methods for load-balancing, all relieved the receive user from overworking.

But the first one leaves the excluded requests alone, and all request users will either be replied or wait for the length of timeout to try another time for reply. Usually, the request user may spend much time waiting just for several tries of the request.

The second one is optimal if the receive user has the power to handle all the requests he receive on time. However, this is a peer-to-peer network; the scale of the network is unlimited, one receive user may have arbitrary requests coming all the time. Only a user with unlimited computational power can handle them all on time but such user obviously does not exist. Thus, in extreme condition when too many requests are received over a long time, this method waste both the request user‘s waiting time and the receive user's computational power.

The third method is to distribute the request between nodes with ability to reply. Hopefully, the group of nodes with same ability to reply is large enough to handle the request in the extension network. Thus, this method is used as a base method in extension network load-balancing. However, more requests are sent over the network during this process and as the most preferable nodes are more likely to be always busy, the average request number needed is several times of the actual request number. The overhead of the network is then increased.

It can be seen that in the above load-balancing models, the request node is assumed to be with not sufficient knowledge to care about the target node. Even if the request node actually have a few not so exact knowledge of the target node, the situation can be much better.

Assume the request user knows the approximate order of how busy all the known nodes are. He can try to send the request to the node with higher preference and less traffic.

By careful calculation, he can select the one node which will most probably reply within the shortest time and send the request to it. This prevents target node from over working and eliminates the extra time of waiting if the calculation is correct.

As the assumption stated, the order of the busy level is only approximate. So though his calculation is 100 percent correct and optimal on the known order, there is still chance that it may fail in practice. This chance is related to the error of the busy level order. The more correct the order is, the better chance the request will success.

The stability parameter gives the extension limited knowledge of such an order, with its correctness probability raised on each update and dropped overtime. This extends the simple methods described in previous paragraphs and leads to more sophisticate load-balancing models described in chapter 3..

1.4 UNRELIABLE ENVIRONMENT ISSUES

Each node in the network can be back traced to one of the Conception instances. Essentially, it is only software that is backing up the whole network. But this software won’t be running all along with the system, and also, the edit period of concept maps won’t last very long. This makes the nodes in the network joins and exits frequently which makes the whole network relatively unstable.

The concurrency control in a reliable environment has been researched thoroughly and has developed mature methods on databases, and the principle of them can be applied to this distributed extension such as Multi-version Concurrency Control, Commitment

Ordering and Strong Strict two-phase Locking. Basically, this extension has a solid research ground to base on, but the unreliable property of the network still is an issue to the implement of the extension.

In this unreliable network environment, each node may disappear or disconnect from the network at any time, which will interrupt even atomic operations. Thus, the specifications below are described as how the atomic operations in this extension are implemented:

- Atomic operations in unreliable environment should be redefined to deal with situations where fatal errors may occur. They are designed to be aware of the failures and most of them are designed with more independency for the ease of recovery.
- In this extension, an atomic operation is limited to a very small operation which should try not to be interrupted by the many unexpected changes in the network. However, each packet transferred over communication contains several actual atomic operations. Both the ability to avoid interruption and the execute efficiency of these operations are taken into consideration
- Inefficient operation which may often cause rollback and long time lock of resources due to unreachable target are divided into smaller operations which are able to recover from these failures. The divisions of these operations not only apply to the operations themselves, the data to be operated on is also divided in some cases. It makes breaking large operations into smaller ones much easier.
- All operations may try to avoid operation failure which involves disconnected nodes by try caching these operations in the network for reconciliation at a later time. The extension does not really cache these operation requests unless the target is confirmed to be disconnected from the network. The caching is more likely to be happening on the more stable nodes.

The extension will try to avoid failures with both relatively low network and local resource cost. The probability of failure remains relatively stable overtime though the stability of the whole network changes within an acceptable range (Approximately 55% to 100%), though there is not much improvement in performance when the average stability is larger than about 75%. Also, if the whole network is totally stable i.e. with an average stability of 100%, overhead is increased over the whole network.

2 EXTENSION & IMPROVED METHOD SPECIFICATIONS

These specifications are used in the design and implement of both the extension and the methods involved. It would be better to give an overview of them before the detailed description of the extension and its working models. Each of these specifications will be further explained in the following chapters.

2.1 GENERAL SPECIFICATIONS

- Concept maps are recognized by the extension as a sequence of Versions added up. Each Version belongs to one of the users and contains several Subversions. Subversion represents one operation on this concept map, such as adding an object, delete an object, and modify an object.
- As the access to the source code of Conception is limited. The demonstration and the testing of this project are done by using the command interpreter as an actual command source.

2.1.1 COMMUNICATION SPECIFICATIONS

- All data to be transferred between nodes is encapsulated into a Packet object. It is then automatically encapsulated into a TCP packet and sent.
- All nodes update their contact list passively, unless a Lookup command is explicitly or implicitly issued.
- Each node do not necessarily wait for response on the same socket; most of the time, after a command packet is sent, it adds itself to the respond waiting list and the extension will wait for it passively until this request timeout.
- In a relayed transmission, the success of this transmission is determined by the final result, i.e. whether the packet is sent successfully to the target. The penalty and raise applied to the stability will according to only the final results, and will be applied to all involved nodes.
- There are two contact lists in the extension, a primary one and a secondary one. The primary contact list contains only the most reliable ones, the size of which is set by configuration, by default, it is 128. The secondary contact list records all other nodes, which is used less often. No contact will be put to secondary list until the primary list is almost full.

2.1.2 STORAGE SPECIFICATIONS

- After one of the users creates a concept map, every user editing it will try to store a full local copy of the versions of that map. Each time a concept map is committed, it is updated to a new version which is then broadcasted across the network. The extension's version control will notify the other nodes of this change. All the involved extension synchronizers will then update their own local copy accordingly.

2.2 COMMITMENT & SYNCHRONIZATION SPECIFICATIONS

- A version header is represented by the corresponding concept map identifier, the current version identifier and the identifier of the user edited it. The command sequence of this edit operation is sent along within the version, capsulated in subversions. The version id is a globally determined serial id and all nodes editing the same concept map should have consent on this value (Otherwise an error happens). While in searching requests, only the map id and version id is used.
- A version history will be stored for each edited concept map to indicate which versions of it have been applied. Thus older version will not accidentally overwrite a newer version. Rollback operation mainly relies on the version history to function correctly.
- To reduce overhead in storage, the number of versions stored in the version history is limited. The version history will cache older versions to external files when the number of history versions stored has reached critical level or when the commitment of a version is permanently confirmed (The editing user has double-confirmed it or the editing user exited).

3 IMPROVED METHODS

Three different techniques (methods) are modified or improved to make them fit for the unreliable extension network. They are the commitment ordering, load-balancing and version control introduced in the chapter 1. The first two are improved by introducing the stability parameter, which is to be discussed in the next few sections. The version control is fundamentally modified and only essential elements of a common version control are retained. It is described in the last section

3.1 STABILITY OF NODE

As stated before, the statuses of nodes in the network are not stable. So the extent of how one node's status can remain unchanged can be measured as a variable. This variable is called the stability of this node. It indicates the ability this node stay available in the network, which is one aspect of the ability to stay unchanged.

The stability value is a probability value, thus, it lies within [0,1]. Also, the correctness of this value has a probability, which can be called the credibility. The credibility is not generally included in this extension, since it adds much additional resource requirements to the whole network. However, at some point where correctness is important and the increase in correctness brings benefits weight more than the cost, credibility is implemented implicitly into the extension.

STABILITY ACCUMULATIVE CALCULATION

Stability changes each time a connection try is made, thus it is better to be accumulatively calculated (*n* is the serial number):

$$Stability(T,n) = \sqrt{Stability(T,n-1) \times \frac{Con_{success}}{Con_{total}} \times \frac{T_{up}}{T_{total}}}$$

T: Target node.

$Con_{success}$: Successful connections made with this node.

Con_{total} : Total connections tried with this node.

T_{up} : Estimated target node up time.

T_{total} : Self running time.

$n = Con_{total}$

$Stability(T, 0) = 0.75$

When the first time a target node is being connected, it can be assumed that the chance of a successful connection is 1/2 because nothing on whether this connection will be successful is known; also, the target node is assumed to be up all along if the connection ended successful, that means $T_{up} = T_{total}$; thus the initial Stability (T, 0) should be equal to 1/2. However, the result of the testing shows it would be better to raise this initial value so that better performance and credibility can be get at the initial phase of the extension execution process..

The value of stability is accumulative. Each time a connection is tried, the result of it adds up to the stability value. And each time the result will be the geometric mean value of the previous stability and the current stability. The use of geometric mean guarantees the stability value drops faster than arithmetic mean.

3.2 COMMITMENT ORDERING WITH STABILITY

The main goal of commitment ordering is to ensure for each pair of operation, one of them is committed before the other one. In some cases in the extension network, two operations won't interfere with each other. In that case, commitment ordering allows the commitment to be made in partial order.

In this extension, the commitment and co-ordinate are handled locally since no global co-coordinator exists. Each of the commitment to be made are required to calculate its weight value, the details are given below:

COMMITMENT ORDERING WEIGHT CALCULATION

$$Weight_{CO}(TS, CR) = \frac{1}{\sqrt{\ln(TS \times \sqrt{CR + 1})}}$$

TS: Timestamp, the time of when the commitment occurs.

CR: Commitment Rate, the count of successful commitments made during a set length of time.

This calculation ensures that:

- A commitment with lower timestamp value i.e. started earlier weights more and actually will be actually committed earlier.

- A commitment with higher commitment rate i.e. commits more often weights less and will be actually committed later. This guarantees the commitment ordering is fair, without staring any committees.
- Timestamp affects weight more than commitment rate, sin it is more important factor to the commitment order.
- The value of the weight is normalized, so it lies within (0, 1) and is represented by a double variable. The formula used guarantees it is not too small, so that the limited precise bits of this double value can be used for correct comparison.
- This weight can be calculated individually by a node, its value will be calculated prior to the pre-commitment phase, which detects and resolves the conflict. While a conflict occurs, both nodes can simply compare their own weight with the other node's weight to determine their commitment order, no further co-ordinate communications are needed. Since both weights are pre-calculated values, the result of both nodes should comply with each other. In the rare case where their weights are exactly the same, they make co-ordinate communications to determine the order.
- The chance of two nodes get the same timestamp for their commitment at the same moment is really small. Also the chance of commitment rate being the same also has a probability less than 1. Thus, the situation where the weight is calculated to be the same value by different nodes over the same period of time is rare. Even when it happens, only a further 2-node co-ordinance is needed to determine the commitment order.

By the nature of this calculation, the weight value is relatively independent as it doesn't require the comparison of two commitments by the time of calculation.

After the value is calculated (within a very short time), it will be sent to other nodes for voting (as the commitment ordering described in [5] requires). If it doesn't cause conflict, a node will immediately vote approve and add the request node to the corresponding commitment queue.

Otherwise, when a conflict happens, the receiver will then compare the weights to determine the order. If the request node commitment weights more, the receiver will act as if no conflict exists, except that the commitment it makes will be suspended until its turn. If a conflict happens, the receiver must have sent its commitment request as well, thus, the same process will happen in the request node and as the weight values are both pre-calculated, they will get the same result for the comparison, that mean one of them will commit first, and they both consent on it.

Normally, a commitment can be made when all votes are collected in the network. However, as the network is unstable, it is possible that some of the votes will never be got. Wait until timeout is apparently not efficient. Thus, the request node will calculate the average stability of all the nodes it has sent voting request to, and determines how many votes have to be got to assume no conflict exists. It is simply using the average stability as the probability to determine this value. Surely, this will lead to failure in some cases, but it is much more efficient than waiting despite the increased cost of failure.

The chance a failure happens is small, as the vote still needs to reach the amount required by the average stability. And then there is another fifty-percent probability that a conflict will not happen. Thus, the actual probability of an omitted conflict is (1-average stability)/2. It will be a very small probability if the average stability is large enough. That is why the extension works well on a network with an average stability that lies within [0.55, 0.75] where the probability of an omitted conflict occurrence is within the range of [0.125, 0.225]. An almost stable network won't be affected by these techniques but will have increased overheads.

3.3 LOAD BALANCING WITH STABILITY

Though the extensions resemble both distributed system and peer-to-peer network, and both concepts have load-balancing methods, the extension will only do load-balancing on the peer-to-peer network.

It is because the design of the extension is totally decentralized. Under this circumstance, a large part of the load-balancing in distributed system is actually the same as that in peer-to-peer network. If the load-balance can be guaranteed in the peer-to-peer network, so can it in the distributed system. There is no need to implement two components overlaps each other.

3.3.1 PEER-TO-PEER NETWORK LOAD BALANCING

In a peer-to-peer network, all communications are started by individual nodes without knowledge of the traffic status of the network. In my extension, they tends to choose the more reliable ones to communicate, this will eventually make the few most stable nodes in the whole network hotspots of communication. In that case, they become super-nodes, whose pattern of communication seems more like that of a server's in the network. The traffic for these nodes will be really busy while much less communication occurs between other nodes. The purpose of load-balancing in peer-to-peer network should be to prevent this from happening.

It would be better if the user can control how much work their node can do before interfering the network traffic. Thus, the user sets the maximum number of inward tasks the extension will handle concurrently. After this number of tasks handled, no more tasks will be accepted. This will help to prevent forming super-nodes, but at the cost of communication failure rate.

However, if this deny-after-maximum method is used together with the stability, the situation will be much better. When a target node refuses to respond, the task will eventually timeout and then the stability of the target node will be lowered (by the accumulative stability calculation formula). Further communication target will be less likely that node (because stability drops fast). After enough communications have been made, the stability will auto-adjust to fit the target node's maximum setting, and the traffic will be distributed more evenly in the whole network.

CONTACT PRIORITY WEIGHT CALCULATION

$$PriorityWeight(T) = StaticPriority(T) + OrderOf(T)/MaxSize$$

T: Target node.

Static priority of a node is normally the same value, which is 0.5. This value is for the user to set to affect the selection of node.

Order of a node is its ordered index in the contact list (primary contact list only). The max size is the length of the contact list.

The priority weight of a node is defined as its dynamic priority when choosing the target node.

CONTACT SELECT WEIGHT CALCULATION

$$Weight_{CS}(T, n) = Stability(T, n) + PriorityWeight(T)$$

T: Target node.

This calculation ensures that:

- A node with higher stability on the next predicted connection will be tried to contact first.
- A node with higher dynamic priority will be selected first.

The node most probability available will be selected to communicate each time.

No nodes in network will overwork because they will refuse excess requests.

As each communication fixes the stability value and increase the credibility of it, a node will be able to better avoid communication failures. Both the failures caused by unstable network and the ones caused by node refuse are tried to be avoided.

Also, this method depends on the average stability of the whole network to work at optimal performance. The range of the stability value determines the range of the weight value. With an overall lower value of stability, the weight is affected much more by the priority weight of the target node, which is a value partially user defined. In that way, the contact select weight is rendered almost useless as it no longer represents the status of the whole network stability. No improvement is got from selecting the contacts according to this weight. In the ideal average stability range of [0.55, 0.75], the stability value contributes more than 50% of the weight value, which makes the weight reflects more of the network status.

3.4 INCREMENT VERSION CONTROL

The distributed concept mapping process can also utilize version control principles. The concept map is the resource being managed. Several users may edit the same concept map simultaneously; each of them is working their branch version of the concept map while the concept map remains to be the unedited base version to others. The commitment operation merges the edited version with the base version in order.

In this extension, the concept of version is a little different. They are represented in an increment way. The main version is the mostly accepted highest version in the network (The committed highest version). The base version is the user's known highest version. A single version does not necessarily represent a whole concept map. Instead, to get the selected version of a concept map, all the versions before the selected version (with a lower version id) is required to reconstruct it.

This is because when commitment happens, the newer version is not physically merged into the older version, only related components are updated. This is enough for the users to feel the newer version is merged. Each of the versions of the concept map is independent from other versions because it only records the changes. Thus, the base version here is not so important and all branch versions are created from nothing without a strict base version.

The figure below shows the process of main version update in the extension network. Branch versions are made from the based main versions, each takes different time editing. However, different than normal version control, if a branch version based on a later main version is finished first, it is committed first (1A is committed as 2 before 0B is committed). Also, the versions with the common sense concept are the boxes on the left side, they the collaboration of the concept map versions.

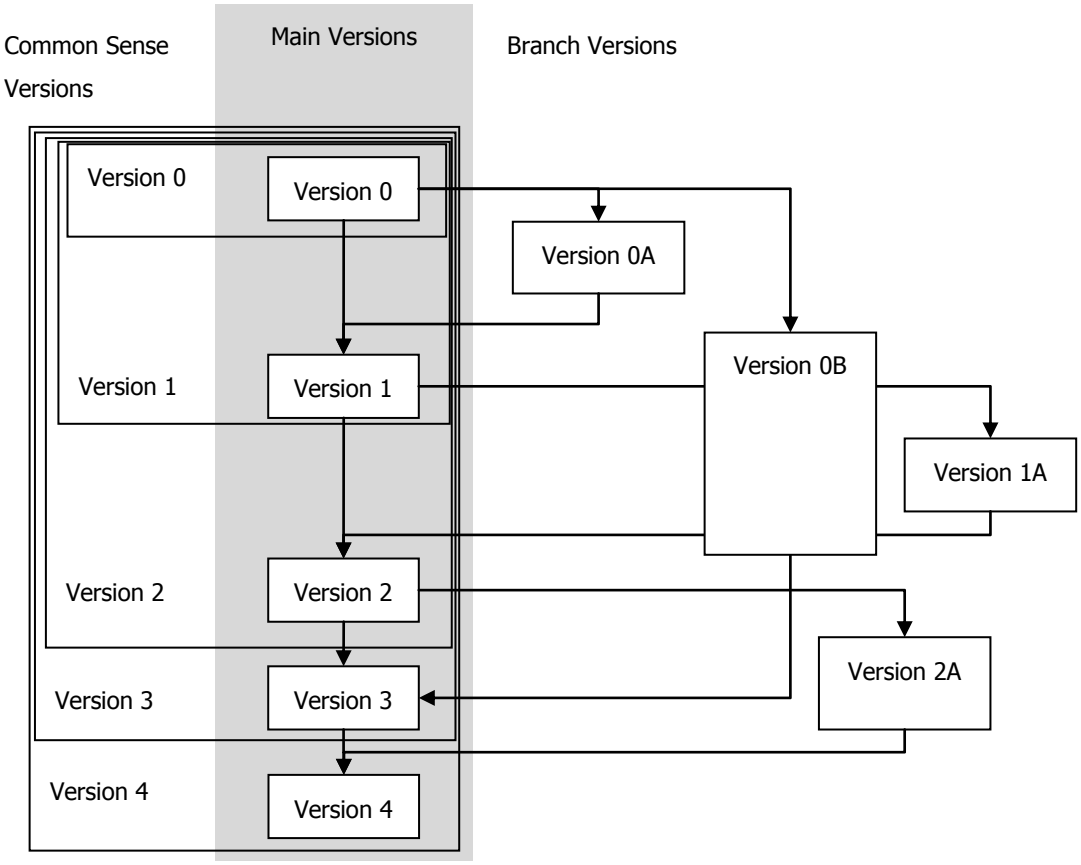

Figure 3-1 Extension Version Control Model

The correctness of such design is guaranteed by controlling the merging operation of branch versions. As any subversion in the branch also has its own timestamp. If a subversion to be applied is conflict with existing ones, the timestamp is checked and the one comes later is applied. The whole sequence of subversions is guaranteed to be ordered by timestamp. Though the sequence of the concept map versions should be kept in order ideally, but it is enough to keep the sequence of subversions in order to keep edit of concept maps consistent.

Due to the nature of concept map, the operations in subversions are largely adding objects. The add object operations don't interfere each other and also don't conflict with the second frequent operation modify object. So, normally, the subversions don't conflict often. (To remove an object within a concept map requires total re-design of all the objects it directly or indirectly relating to, there is large amount of work to do for the user, so it is really rare that a remove operation will appear. Instead, to remove an object, normally the user modifies the corresponding object as an obsolete object.)

This design can make rollback operations less complex and less expensive. Rollback **i**s also needed less. Local subversion timestamp check normally fixes most wrongly committed versions. Sometimes a wrongly committed version only needs its id fixed and the effects recommitted; each node will notice the problem because the next few commitments will have an incorrect version id (not the next integer of the current version). They will just fix it on their own by checking the right version sequence. Thus, a rollback is avoided and the problem is fixed eventually without causing much performance reduction.

4 SOFTWARE DESIGN

4.1 SOURCE CODE PACKAGE AND CLASS HIERARCHY

The hierarchy of the source code is shown in this part for the ease of linking the abstract content and the actual file together in this chapter.

All source codes are written in Java using 32-bit version of Eclipse Galileo. Visual Editor plug-in is used for the ease of user interface design. Only original packages of Sun JDK 1.6.0.21 are used in this extension, no third-party library is used.

All source code are written by myself except the few lines generated by the Eclipse environment.

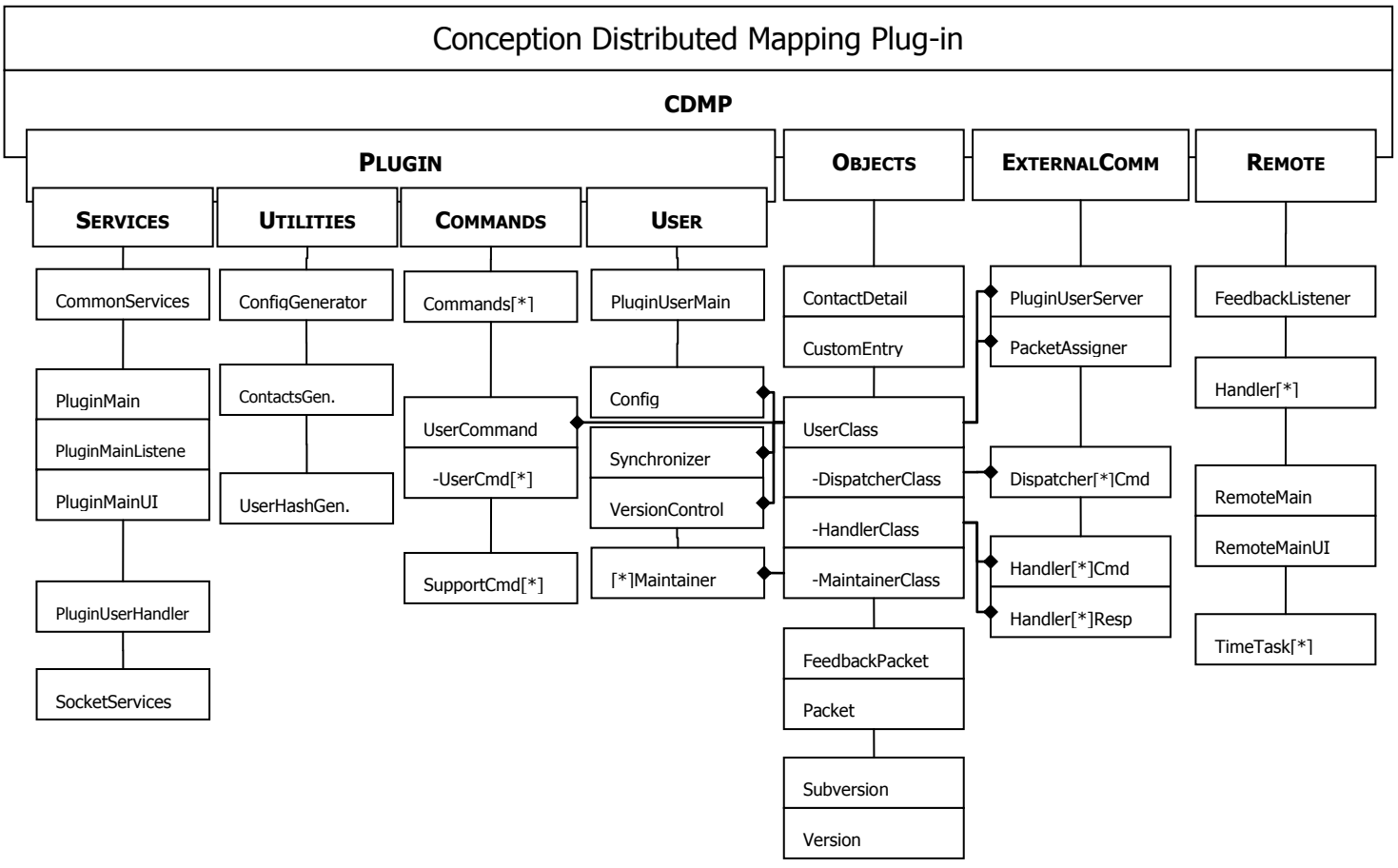

Figure 4-1 Source code hierarchy

[*] represents similar classes with the same superclass, [*] is the can be replaced by any string.

Vertical Links in the same column links classes in the same package together.

Classes in the same column being shown in a cascade pattern represents inherence relations. The first one in row is the super class and the other classes under it starts with " -" are the sub-classes.

Horizontal Links between classes also represents inherence relations, which happen between classes in different packages. The diamond shaped end indicates the sub-classes.

The whole extension is placed into the package of cdmp, which is the abbreviation of the extension's name.

The package cdmp.Plugin is the core of the extension, and the package cdmp.ExternalComm contains all the classes dealing with communication. The abstract objects used are put into the package cdmp.Objects. All user-side resource is put into cdmp.Remote package.

4.2 USER AND EXTENSION

The concept of the user and the extension are used in different forms when viewed from different perspectives.

User thread source code: cdmp.RemoteUser.RemoteMain

When describing the physical extension network, user thread will also be called RemoteMain.

When describing the logical extension network, it will be called node.

The term user is used for both circumstances.

Extension main thread source code: cdmp.PluginServices.PluginMain

When describing the physical extension network, extension main thread will also be called PluginMain.

When describing the logical extension network, it will not be mentioned explicitly since it doesn't belong to the logical network.

Nonetheless, the term is used for both physical and logical networks when even though it is not directly related to the logical network.

4.3 WORKING MODES

The extension is designed to work in two modes: Single-user Mode and Multi-user Mode. Multi-user mode allows several users remotely use this extension instance as if they have their own extension running locally. Multi-user mode extension does not share resource among its users, each user is logically independent.

4.3.1 SINGLE-USER MODE

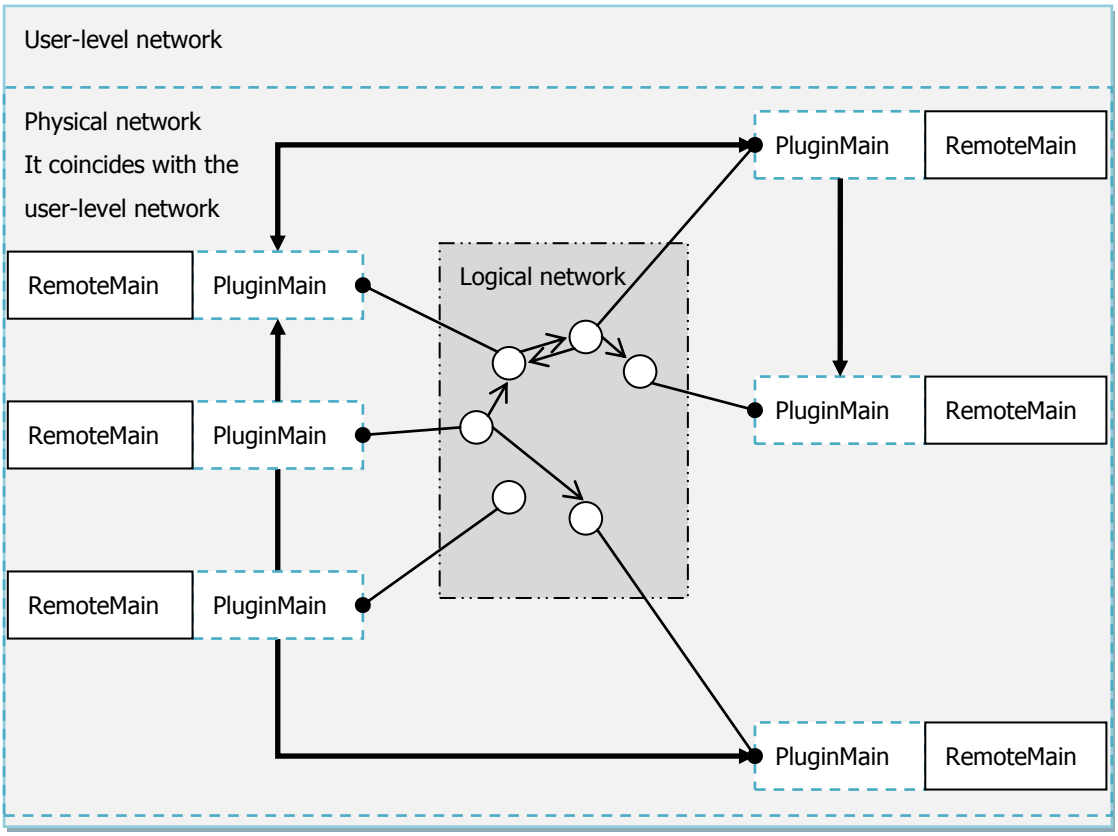

Figure 4-2 Single-user mode extension networks

Extension main thread only accepts one user who is working locally. No user listener will keep running after the connection of local user is established. The extension exits with this user.

In this mode, the RemoteMain(user) and the PluinMain(extension) are coupled.

4.3.2 *MULTI-USER MODE*

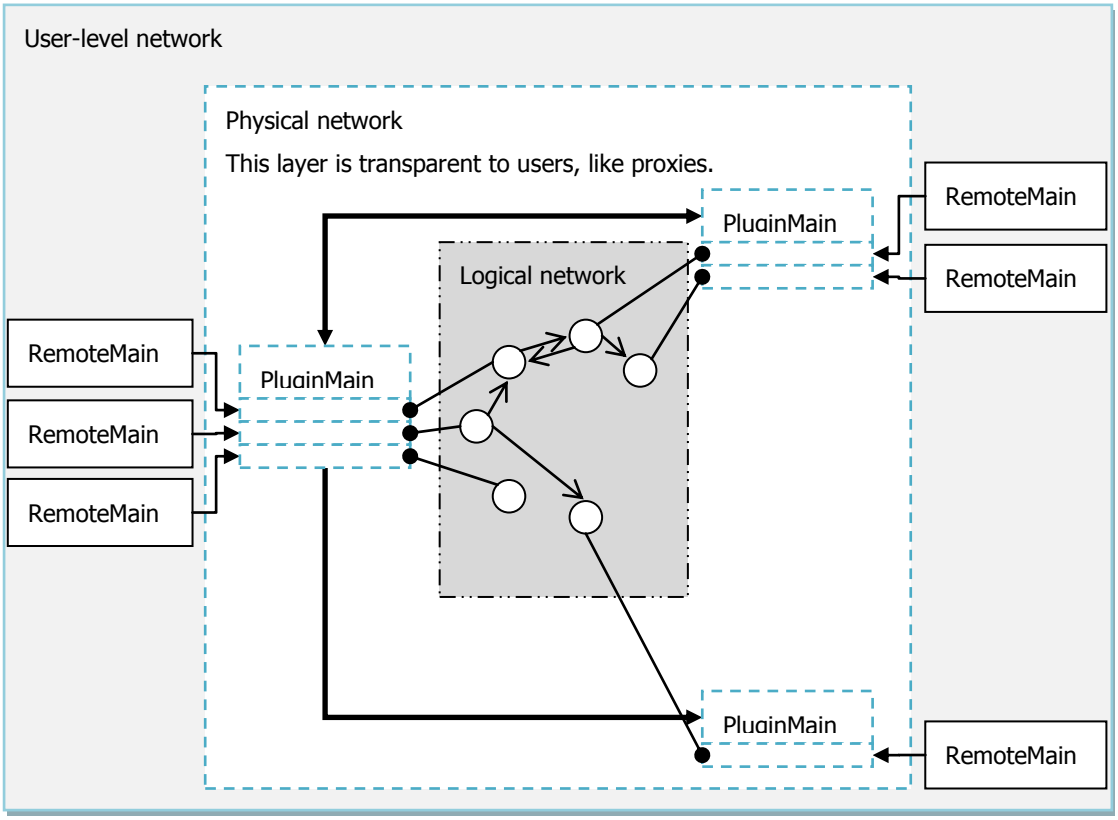

Figure 4-3 Multi-user mode extension networks

In multi-user mode, several users can use the same extension. An instance of user thread is allocated to each user, which has the same functionality as a single-user mode extension

Physically, RemoteMain(users) are connected to the PluginMain(extension), which process the data for each user and do the actual work.

However, the PluginMain is not actually an element of the logical network, only the RemoteMains, which is seen as nodes or contacts are elements of the network. The layer of which all actual users connect to is transparent to the users, they acts as if they are directly communicating with other nodes.

Logically, only users (RemoteMain) exists in this network as its nodes.

4.4 ARCHITECTURE (SINGLE-USER MODE EXTENSION)

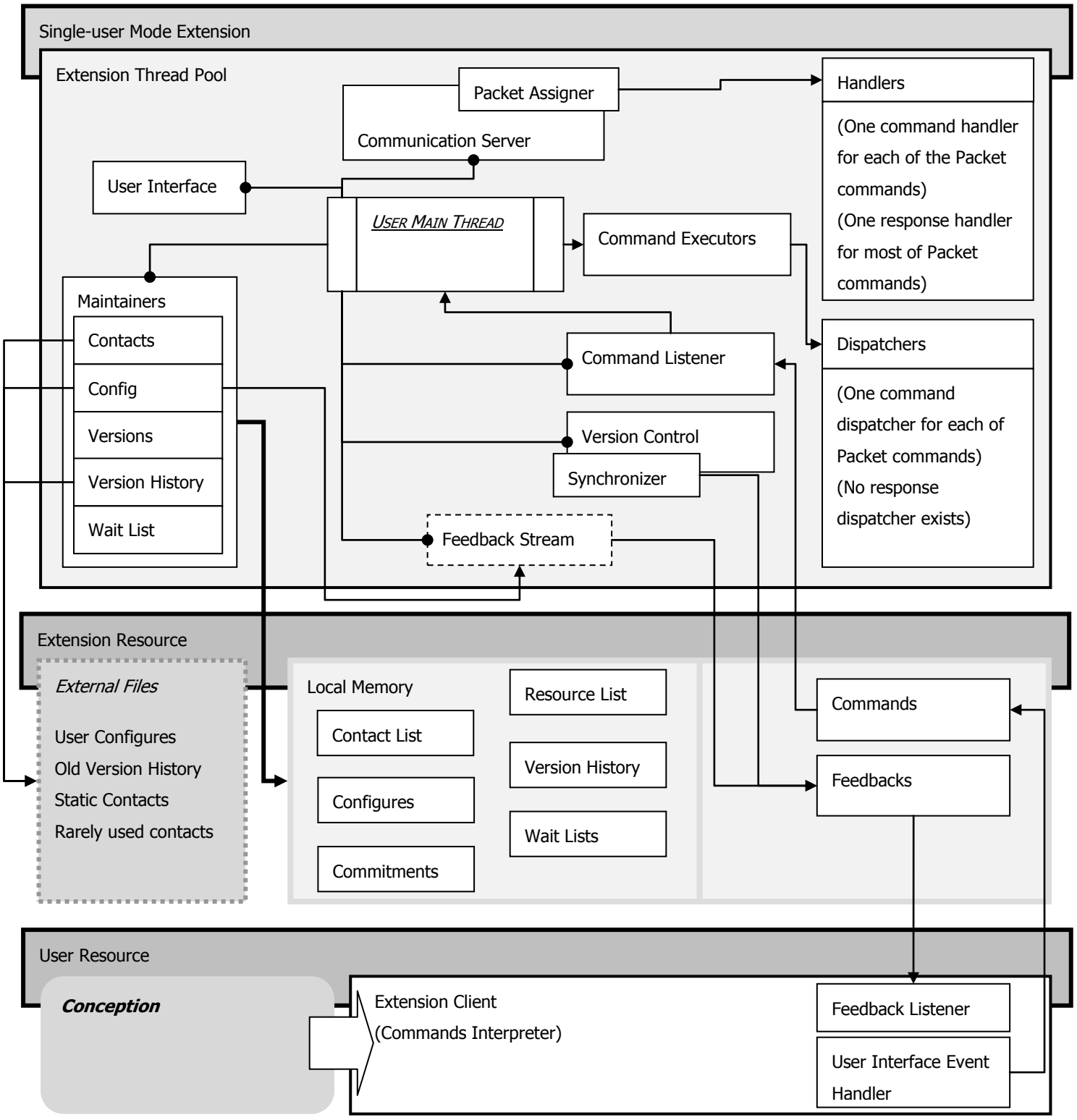

Figure 4-4 Single-user mode extension architecture

Except the instance of Conception, and the abstract object Feedback Stream, all other objects in this figure represent actual parts implemented in the extension source code.

The whole implementation is a bit complex to describe, thus, the details of its working processes are listed in the software functionality section at the end of this chapter in a more independent pattern. Only an overview of the extension functionality will be given here.

The whole architecture is divided into three layers. On the top is the threads and main context of the extension. At the bottom are the actual user interfaces and it is where all the work flows will start from. In the middle are the resource used by both the extension and the client. The left part of the local memory is the runtime resource used by the extension, and the right part is the data shared and transferred between the extension and the client. The external files listed all the files needed in runtime, however, some other files also exists to persistently store data accumulated in runtime, such as the contact list file (records the primary contact list), the resource list file (records resource of self and primary contacts), and the operation caching file (record interrupted operations).

The working flow starts from the lower-left corner, from the Conception. The operations of the user are directly given to extension client, which can also be seen as a command interpreter, depending on how it is used. If the commands are all issued from the Conception, it is considered only an interpreter. However, the same commands can also be issued from the client interface using raw data input. In this way, it is being used as an extension client. (It is not necessarily linked to a Conception instance directly when used like this. The tests in this project are done in this way.)

These operations (or commands) are then transferred to the extension along with their parameters by the event handlers. The effect of a command from a Conception instance is the same as one issued from the buttons on the user interface. The data is not explicitly encapsulated during this transmission, after the command identifier is sent, all parameters are sent one by one according to the order consented by both the client and the extension.

The command data is caught by the extension’s command listener. It translates the general command to its exact type and tries to get the parameters accordingly. If all required data is got, the listener will request the user main thread to start the corresponding command executor.

The command executors contain the exact code to be executed for the commands. For commands need communication, the command executor will start dispatchers to do

the communication. Otherwise, the command executor will be the end point of the work flow.

All out-ward communications must be sent by one of the dispatchers. And all in-ward communications must be received by the communication server. There are two kinds of possible in-ward communications, command and response. The communication server doesn't check the content of the received packet. All packet will be checked by a packet assigner, first is the security and integrity check, and then a handler thread will be started to actually work on the packet content. Different type of packet will have different handler. The handler is the end point of work flow for almost all response packets and for those commands that don't need response. Otherwise, additional dispatchers will be started to either send a respond or a command. The same process will then happen in the target extension. The communication will continue until the work flow reaches its end-point.

During all the processes, there will be messages generated to notify the user of the progress and the result. They are transferred back to the extension client using the feedback stream. The data is encapsulated into a feedback packet. The feedback listener catches them and displays the messages. The commands to be executed by the client (and the Conception) are also transferred in this way, the listener will start client handler to deal with them. Currently, the only command to be expected is the commit new version command from the synchronizer.

The maintainers are essential for the extension. They don't belong to the user working flow, but they maintain the data and parameters to be used in the working flow. Work such as updating the stability parameters, ordering the contacts in contact list and record the version commitment history are done by the maintainers. Also, the data they maintain resides in the corresponding maintainers' thread.

External files are used by the maintainer for additional storage or for persistent data storage. User configures is the persistent storage of config. Static contact file is edited externally (by the contact generator in this project). The content is to be loaded every time the extension starts. The old version history provides both additional and persistent storage of the whole version commit history. Rarely used contacts are only temporary file created to store the excess contacts in secondary contact list. Normally, they will not be used at all, as they are the most unstable ones.

The user interface of the extension is used to record all the command processed and to monitor its working status. Messages of command received are mainly the content displayed. Most critical failure messages will also be displayed on it.

4.5 ARCHITECTURE (MULTI-USER MODE EXTENSION)

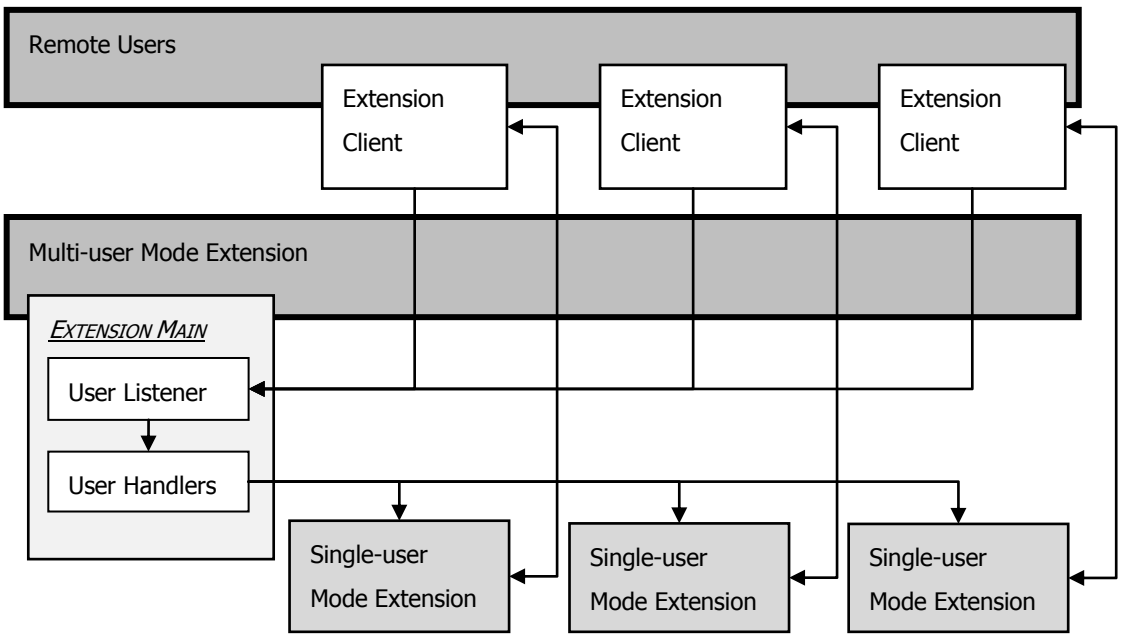

Figure 4-5 Multi-user mode extension architecture

This multi-user mode architecture is actually based upon the single-user mode architecture. Several single-user mode extensions are combined into a single extension. Additional threads are used for multi-user control. Most of all, the central thread of the extension is changed to the extension main thread. This means the entry point of the extension is different for different modes.

The process is simple. After the extension starts, only its core thread and a user listener is started. It waits until any of the clients send a connect request. The listener will simply check if this user exists, and then start a user handler to do the remaining job.

The user handler will do further checking on the request user and if the user request is valid (in security and format), the user handler will start a single-user mode sub-extension by creating new user main thread. There will be short communications between the handler and the client to confirm their out-going port and internal communication sockets. After the sub-extension fully starts, the handler will exist and the client will directly send command and receive feedback from the sub-extension as same as using a single-user extension.

4.6 ABSTRACT OBJECTS & COMMANDS

4.6.1 USER HASH

User hash is a hexadecimal string with a length of 32, and is used as the main part of the identifier of a user. (Other parts are the ip and port, which can be dynamic between each runs)

It is defined and obtained as the following:

- Obtain the local host and append the first 8 bits of its java hash code (transformed to a hexadecimal string) to the result. If local host doesn't exist, use the system properties' hash code (transformed to a hexadecimal string) instead.
- Append the first 8 bits of the current java runtime's hash code (transformed to a hexadecimal string) to the result.
- Fill the result to a length of 32 with randomly generated hex numbers.

The hash function used is the native java hashCode function. It is not strictly a hash function, however. But it is good enough to create non-collision numbers if the contexts used are non-collision. All contexts used (the local host object, the system properties object and the java runtime object) are normally represented differently if on different computers, the chance that both of the first 8-bit and the second 8-bit are the same is very small. Also, the last 16-bits are generated randomly. This makes it possible to use several clients on one computer without causing collision. The random factor also make the chance of collision smaller.

Implemented in cdmp.Remote.ReomteMain.

(An alternative user hash generation method is also implemented to generate serial integers as user hash. The alternative method is used for testing because a short unique user id will be clearer than a 32-bit hexadecimal string.)

4.6.2 CONTACT

In the node's perspective, nodes which can be reached within reasonable delay are considered its contacts.

A contact is mainly stored within a ContactDetail object, which record the information needed for communication. However, its id is stored in the contact list as the key value to access them.

The ContactDetail object contains: the count of out-ward and in-ward communications, the contact's average response time, the last known available time of this contact, the stability and the stability factor of the contact and a active flag.

All the information in ContactDetail object are mostly used to calculate the stability value, except the active flag, which indicates whether this contact disconnected physically.

Implemented in cdmp.Objects.ContactDetail.

4.6.3 SUBVERSION & COMMANDS

A subversion represents and stores the information of one edit operation on a concept map. A subversion stores the id of the concept map edited, the operation command, the object involved and a timestamp for this edit operation.

The operations are defined and limited by the subversion commands listed below. Each of them conflicts some of the other operations. The order of the operations is important to determine a conflict. The conflict operation listed represent this operation will cause a conflict if the one of the operations listed was the last operation committed to the same object.

SUBVERSION COMMANDS: ADD

Conflicts: Remove

SUBVERSION COMMANDS: REMOVE

Conflicts: Add, Remove, Modify (Normally everything)

SUBVERSION COMMANDS: MODIFY

Conflicts: Remove, Modify

Conflict is only detected when a subversion is to be committed in an unexpected way, such as late commit or re-commit. In normal commit process, conflict operations will be automatically override by the newer operations.

Subversion implemented in cdmp.Objects.Subversion.

Its command enumerated in cdmp.Plugin.Commands.CommandsSubversion.

4.6.4 VERSION

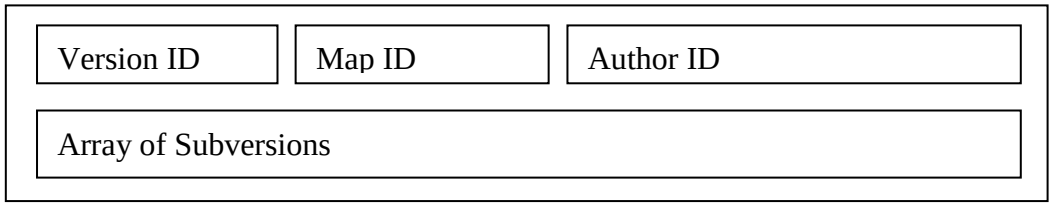

Figure 4-6 Structure of a Version

A version consists of a globally serialized version ID, the ID of the map it contains, the ID of the author, and the actual content of this version, the list of Subversions.

Version is the context to be committed. The version id is not determined when a new version is assembled; it will be assigned an id when all other contacts approve it in commitment ordering.

The commitment of a Version is actually done by commit the Subversions it contains. It is simply used to store the Subversions and also used as an abstract context in commitment ordering.

Implemented in cdmp.Objects.Version.

4.6.5 VERSION HISTORY

Version history records the trace of Versions committed. It is used by the rollback operation.

The version history is indexed and newly committed version is appends to the end of it. It has no actual limit on the length but the version history maintainer will try to keep the history at a reasonable length.

Double-confirmed Versions and the Versions with an unavailable author will be kept out of the active version history, and stored into a version history file. The remaining part of the version history will be stored to file when the user exists.

Implemented in cdmp.Plugin.User.VersionHistoryMaintainer.

4.6.6 USER COMMANDS

These commands are used by the client and sent to the extension to take effects. All these commands represent one of the functionality of the client, which are to be executed by the extension.

USER COMMANDS: EXIT

Shutdown this user's thread pool, dispose this user's all resource, and exit the user main thread.

If the extension is working in single-user mode, the extension main thread is interrupted and terminated as well.

USER COMMANDS: LOOKUP

Ask the extension to dispatch lookup packet to all known contacts and update the contact list by the responses received.

This command is usually issued when the user wants to update their knowledge of the structure of the whole network. It is expensive operation and causes much overhead; however, the more accurate the extension knows about the whole network, the better efficiency can be achieved by the extension itself.

USER COMMANDS: SUBVERSION

Send one edit operation on a concept map to the extension.

The operation is to be stored as a subversion. And the subversion will be put into the subversion queue, which are waiting to be assembled as a version.

USER COMMANDS: ADD CONTACT

Send the address information of a contact to the extension to try contact it.

If the contact is successfully connected, the contact will be add to contact list.

USER COMMANDS: VERSION COMMIT

Try committing the version assembled by all the subversions in the extension subversion queue.

If the subversion queue is empty, this command takes no effect, not even lock the status of committing lock of the concept map.

USER COMMANDS: VERSION UPDATE

Request the extension to update the concept map to the latest version.

This command is also executed period to keep the concept map version correct. However, user may want to make sure it is correct at anytime, this command is used in that case. This command is similar to the operation of refreshing a web-page.

User Commands enumerated in cdmp.Plugin.Commands.CommandsUser.

4.6.7 PACKET & COMMANDS

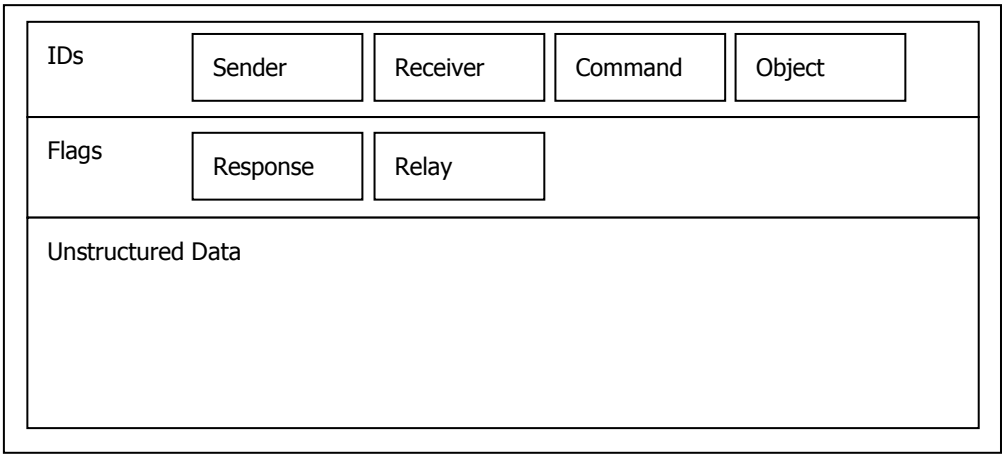

Figure 4-7 Structure of a Packet

A Packet is used to encapsulate a command request or a command response. It records the sender's id and the receiver's id to ensure the correctness of this transmission. The command to be executed and the object it works on also have an id reference recorded.

The response flag is used to indicate this is a response for the corresponding command.

The relay flag is used to indicate this packet is important and it requires the relay of this packet if it can't reach the target directly.

The unstructured data part is used to store additional parameters and information required by this command in a pre-defined order.

The commands of a Packet are listed below:

PACKET COMMANDS: LOOKUP

Internal abbreviation: LKUP

Handles: Automatic request or user issued request for looking up new nodes and updating contact list.

PACKET COMMANDS: VERSION PRE-COMMIT

Internal abbreviation: VPCT

Handles: Automatic request invoked by user issued version commit commands. It is actually where how the commitment ordering takes effect.

PACKET COMMANDS: VERSION COMMIT

Internal abbreviation: VCMT

Handles: Automatic request invoked by user issued version commit commands. This request won't be invoked until it got its order from commitment ordering after the version pre-commit request.

PACKET COMMANDS: VERSION REQUEST

Internal abbreviation: VREQ

Handles: Automatic request or user issued request for updating a concept map to the latest version. Invokes version query command first.

PACKET COMMANDS: VERSION QUERY

Internal abbreviation: VRQR

Handles: Automatic request made by the extension to determine the correct newest version id.

PACKET COMMANDS: RELAY

Internal abbreviation: RLAY

Handles: Explicit relay request of the extension. It is like setting the relay flag true, however, the receiver node will take much more effort to try to send this packet to the target.

All packet commands represent one of the extension functionality, which requires communication using packets.

Packet Implemented in cdmp.Objects.Packet

Packet Commands enumerated in cdmp.Plugin.Commands.CommandsPacket

4.6.8 FEEDBACK PACKET & COMMANDS

The feedback packet is similar to a packet, except that it is used for communications between the extension and the client.

The commands used by feedback packet are listed below:

FEEDBACK PACKET COMMANDS: MESSAGE

This packet contains only a string to be displayed on the client user interface.

FEEDBACK PACKET COMMANDS: VERSION COMMIT

The version confirmed to be committed is send back to the client to update its display and content.

Feedback Commands enumerated in cdmp.Plugin.Commands.CommandsFeedback.

4.7 USER INTERFACE

Both the extension and the client have a user interface of their own. Though both are only used for supplement control and for status monitor of the extension.

The extension UI is simply a monitor of the extension working progress.

The client UI monitors the working progress of its commands, which can also be issued from the UI.

4.7.1 EXTENSION UI

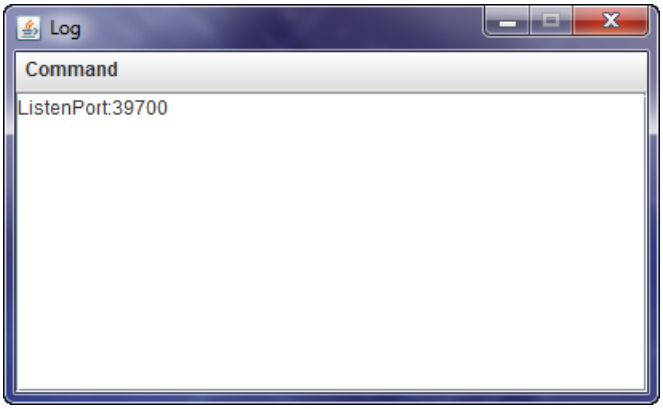

Figure 4-8 Extension User Interface snapshot 1

The interface of the extension is very simple since all commands will be issued from the actual software Conception. However, currently, all commands are issued by the user through another interface.

The whole interface is actually filled by a text area. All the logged messages relating to the extension are displayed here. Also the Command menu bar now only contains one command: exit. The extension won't be closed simply by pressing the cross on the title bar, because it can be easily pressed accidentally. Only by the exit command in the menu bar will this extension exits. (Or by the user command of exit if in single-user mode.)

However, closing the extension through the extension exit command is not recommended when users are connected. Since this will call the shutdown method on all the existing ExecutorServices in the extension. The process of user commands will be interrupted without caching. The user data may be recorded incorrectly or with loss. And though the user will be notified, a socket reset exception will always happen.

This picture of the interface is running in multi-user mode. Thus, it only initializes the user listener and a few other core threads. The port used by the user listener is set by the configure file of the extension, which is loaded during startup. However, the default port is set to 39700 in the source code. When applying for the listening port (server socket), several ports will be tried to bind if the default port is unavailable, until success. Thus, extensions can work on the same computer using same set of configure and storage locations but with different communication ports.

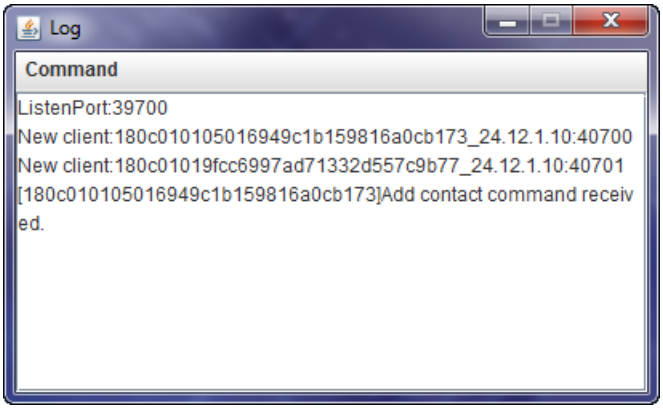

Figure 4-9 Extension User Interface snapshot 2

After one user tries to connect, the extension will do several checks before approving a user to use this extension. And the further work of establishing the user connection will be handled by a sub-thread of the listener.

However, the messages during these session are not output to this log text area because too much detailed messages will make this log hard to read (Though it is still not easy to read at the current state). A full detailed log (which includes the messages sent to the users) can be fetched by redirecting the standard output stream to a file.

Whenever a new user's connect request is approved and its user thread is ready to work, the New client: [User Hash]_[User Communication Address] message will be recorded. In the above picture, two users have successfully connected.

Whenever a user issues a command request, the message [User Hash][Command Name]command received will be displayed. In the above picture an Add Contact command is received from the first connected user.

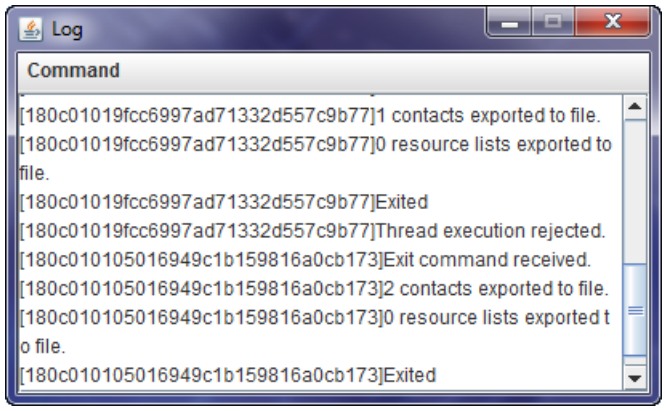

Figure 4-10 Extension User Interface snapshot 3

After one user exits, the thread executor it uses will be shutdown and any further commands will be rejected (A command from user […c9b77] is rejected after it exits in the picture). Its related resource and configure will be exported to files (Only the number of contact and resource list exported will be displayed, other files such as the configure file doesn't generate messages).

4.7.2 CLIENT UI

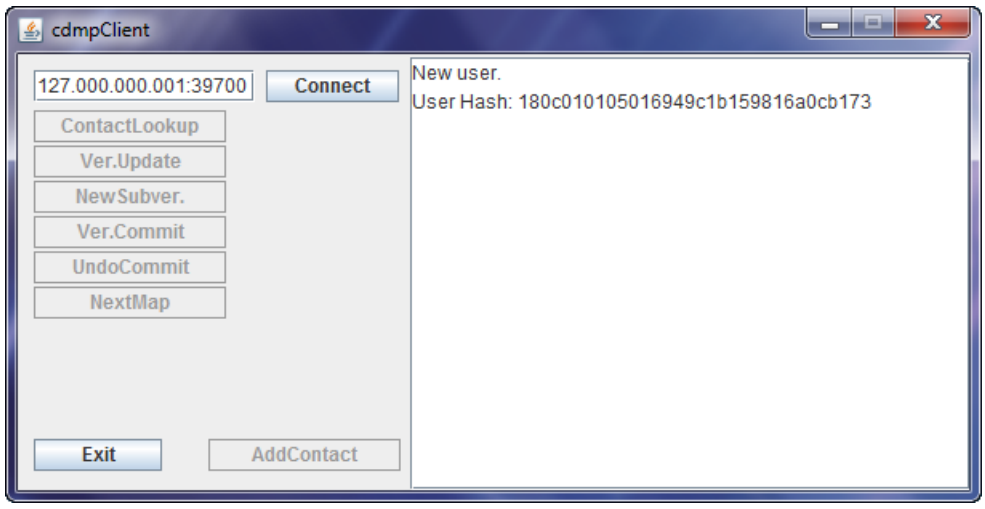

Figure 4-11 Client User Interface snapshot 1

The largest component of the client user interface is also a text area for logging and displaying messages. It occupies the right half of the UI.

The command buttons are all positioned on the right half. Only two buttons are enabled right after the start of the client.

The text field and button at the top-left are used to connect this client to an extension. The text field is used to fill-in the address of the extension. It is displaying the default value of 127.000.000.001:39700, which is the address of an extension running on the local host using the default listening port. In the single-user mode, this extension is auto-started and auto-connected.

The exit button is used to only terminate the client thread when it is not connected. This command is also sent to the extension if connected.

During the start of the client, the client will check if there is an existing user file. On the first run, the client will identify this is the first run since no previous user exists. It will generate a new user hash for the user. They are then displayed on the interface.

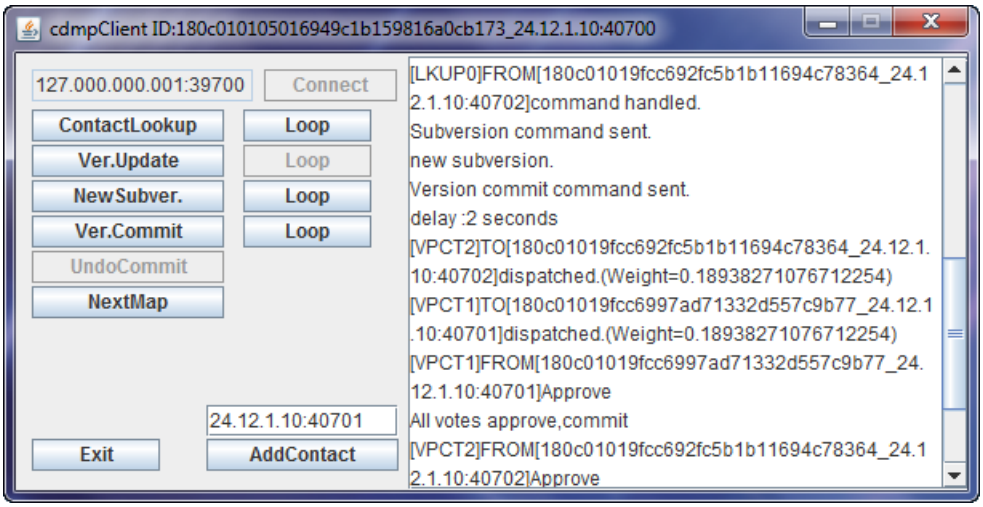

Figure 4-12 Client User Interface snapshot 2

After the client successfully connects to an extension, the connect controls are disabled and the input extension address is displayed as a label on the UI.

Most disabled buttons will be enabled and additional controls will be shown after the connection.

Each of these buttons is used to issue one of the user commands. The Loop buttons is used to issue the commands periodically.

Many different messages are displayed on the UI. Every time a command is sent, the message "[Command Name]command sent" will be displayed. Every time a feedback is received, the message "[Command Abbr.][Transmission Direction][User Hash]status" will be displayed. Additional messages will be displayed as required.

4.8 SOFTWARE FUNCTIONALITY

The functionality of the whole implementation is divided into two sections: the user functionality and the extension functionality.

Each of the functionality will have a chance to trigger failures belonging to one or more categories in the next section.

4.8.1 CATEGORY OF FAILURES SEVERITY

When a failure happens, depending on the exact context of the failure, it can be classified into one of the following categories, each has different occurrence probabilities and resolve methods:

- CRITICAL: The effect of failure is almost permanent in that they will always halt the normal execution of further operations over part of the network. The scale of the network needs to be suspended is considered to the critical level. Higher critical level failure means it is harder to recover from it, and more harsh recovery operations are required.

  This category is avoided by extension at all cost because it can totally mess up the user experience of the extension.

- GLOBAL (REPEATABLE OR RECOVERABLE): These failures still involves large parts of the whole network and requires both extra resource and globally co-ordinance to fix the errors. Though normal flow of operations is not interrupted, it is still delayed by the extra resource cost of repeated operations or recovery operations.

  This category is acceptable since it is recoverable.

- INDIVIDUAL (REPEATABLE OR RECOVERABLE): These failures are similar to the previous category but it only requires extra resource cost locally in the affect nodes. Most of the time, users will have excess local resource because this is not an extension with heavy local workload.

  Preferably, most failures occurred can be put into this category.

- IMPLICIT (REPEATABLE OR RECOVERABLE): These failures are fixed without explicitly invoke recovery or repeat operations. The design of internal mechanism of the functionality takes them into consideration and fixes them internally within normal flow of operations. Very little extra resource is used in the process.

Ideally, failures can be reduced to this category.

- SAFE: The cost to recover the failure can be omitted. These are not considered to be actual failures and most of them should be fixed eventually.

All functionality failure occurrence will be classified into one the above categories.

4.8.2 CLIENT (USER) FUNCTIONALITY

STARTUP INITIALIZATIONS

Trigger event: Upon startup of the user interface. Automatic

Failure Severity: Individual,Implicit

Description: After normal application startup, it will detect whether a local user already exists by checking if the file with both a specific name and valid structure exists. The content of the file will be recognized as the user's user hash, which identifies the user. Otherwise, if the no existing user hash can be get. The generation of a new user hash will be invoked.

CONNECT

Trigger event: By command issued through Connect button. Or triggered automatically in single-user mode.

Failure Severity: Individual,Implicit,Safe

Description: Try to communicate with the remote extension with the address specified in the address text field (On the left of the Connect button).

If the target extension responds, send the user hash and request the extension to allocate resource for this user. (Extension may deny the request if a user with the same user hash is already allocated resource.)

If the request is approved, several more communications are needed to:

Determine the external communicate port to be used.

Establish an additional feedback socket to receive the result and commands from extension while the original socket is used to transfer the user commands.

Wait until the user thread on the extension is ready to work.

After the confirmation of the extension, this user is connected to both the extension and the network.

ADD CONTACT

Trigger event: By command issued through Add Contact button.

Failure Severity: Implicit,Safe

Description: Request the user thread to lookup node at the address given in the contact address text field. The address of the target node should be in this format: "[target node communication IP]:[target node communication port]". The format of the input is not checked. The internal method actually only split the input string by the character ":", treat the first sub-string as the ip and the second string as the port number. If any error happens (mostly because of an invalid input), the request is ignored.

Comment: As the extension won't blindly scan the network for a valid contact node, at least one contact should be added manually at the first run. To guarantee the whole network is connected, several key contacts are added by each extension when a user connects. These key contacts are stored in the static contact list in the extension configuration. The startup contacts are similar in concept to the bootstrap nodes in a KAD network.

REQUEST CONTACT LOOKUP

Trigger event: By command issued through ContactLookup button. Or by the automatic periodic lookup command

Failure Severity: Individual,Implicit,Safe

Description: Request the user thread to initial a lookup command. All the nodes in the primary contact list will be sent a lookup packet. If the primary list is empty, it can only mean this user is isolated and needs to add startup contacts manually before using this command.

Comment: This request would do nothing if this node knows no other nodes. This request would have no effect if all nodes known are unavailable.

REQUEST CACHE EDIT

Trigger event: By command issued through the New Subver. Button. Or when the concept map is saved in Conception.

Failure Severity: Individual,Implicit

Description: Only one edit operation on the concept map is cache by each request. The edit operation is represented by its operation command, object id and map id. The

information is put into a Subversion object and then put into that concept map's subversion queue. Waiting to be committed or discarded.

REQUEST REFRESH CURRENT MAP

Trigger event: By the command issued through the Ver.Update button. Or by the automatic periodic update command.

Failure Severity: Individual,Implicit

Description: Queries for the highest version id of the currently editing concept map will be sent to several nodes. All the primary nodes with this resource will be included. If too few nodes are included, the content of the secondary node list will also be used.

If the local version is lower than the highest version, then sent out all the needed version update requests to the responding nodes to update the content.

REQUEST COMMIT EDIT

Trigger event: By the command issued through the Vers.Commit button.

Failure Severity: Global,Individual,Implicit

Description: Request the extension to assemble and commit a new version. The latter part of this command is done by the extension.

If no subversion can be used to assemble version, this command has no effect.

REQUEST UNDO COMMIT EDIT

Trigger event: By command issued through the UndoCommit button. Or automatically issued when the process of commitment is interrupted.

Failure Severity: Critical,Global,Individual

Description: Request the extension to cancel the process of version commitment. Depending on the progress of the commitment, the operations needed varies.

If the commitment has not been sent to other nodes, the operation will be interrupted right away, no more operations are needed.

If the commitment is waiting in the global queue, a fake commit packet will need to be sent to cancel the commitment.

If the commitment has already been committed, this will require a full rollback, which will have a small chance to cause critical failure (when most subversions of the version committed conflicts the existing ones).

EXIT

Trigger event: By command issued through the Exit button.

Failure Severity: Implicit,Safe

Description: Terminates the client and exit. Also ends the extension process in single-user mode.

4.8.3 EXTENSION FUNCTIONALITY

CONTACT LOOKUP

Trigger event: Startup of extension, by user command add contact or user command contact lookup.

Failure Severity: Implicit,Safe

Description: Start one dispatcher for each contact in the primary list, send lookup packets to these contacts and record this communication in the waiting list then wait for the response.

ASSEMBLE SUBVERSION

Trigger event: By user command new subversion.

Failure Severity: Implicit

Description: The latter part of the user functionality - request cache edit. Store the operation information received in a new subversion object. Then store the subversion in the waiting queue.

ASSEMBLE VERSION

Trigger event: Automatically triggered in the execute process of the user command version commit.

Failure Severity: Implicit,Safe

Description: Gathers all the subversions in the waiting queue, and put them into a new version object. If the queue is empty, notify the user.

COMMIT VERSION

Trigger event: By user command request commit edit.

Failure Severity: Critical,Global,Individual

Description: Trigger the assemble version functionality to get aversion. Initialize the commitment of that version. Then start the functionality of pre-commit. The process of the version commit is suspended until the pre-commit version command ends or when it reaches timeout. If a version id is successfully assigned, dispatchers will be started to send the version to other nodes. Otherwise, the process will have to be retried.

PRE-COMMIT VERSION

Trigger event: support operation for version commitment, triggered automatically.

Failure Severity: Global

Description: Try co-ordinate the operation of version commitment with other nodes. A globally serialized version id is assigned to the version to be committed.

RELAY PACKET

Trigger event: failure of the transmission of an important packet (such as version pre-commit packet) which has relay flag set to true.

Failure Severity: Critical,Global

Description: Packet relay is a rather expensive operation in most cases; it usually takes a relatively long chain of nodes for one delivery. In the worst case, if the target is unavailable in the network, each node on this chain will have to make several tries in vain before notice that.

VERSION QUERY

Trigger event: Automatically triggered during the execution of user command version update.

Failure Severity: Global,Individual

Description: Ask other nodes about the latest committed version id.

VERSION REQUEST

Trigger event: Automatically triggered during the execution of user command version update.

Failure Severity: Global,Individual

Description: Request the target node of the corresponding version. Each missing version will trigger this functionality once in order to obtain it.

EXIT

Trigger event: By command issued through extension menu or by the command of the user.

Failure Severity: Safe

Description: Terminate the extension and release all resource. All connected users will be notified and forced to disconnect.

5 TEST AND EVALUATION

5.1 FUNCTIONALITY TESTS

Each of the extension and client functionality is tested after it is implemented. Most of them functions properly with reasonable efficiency. However, as the whole implementation is quite large and is only implemented and tested within about 3 months by myself. Thus, there are still many errors in it which has not been fixed.

The results are listed in the following sections.

5.1.1 CLIENT (USER) FUNCTIONALITY

STARTUP INITIALIZATIONS

Result: Normal. No internal error found.

CONNECT

Result: Normal. No internal error found.

ADD CONTACT

Result: Normal. An invalid input where the ip and port is invalid but spitted with ":" will cause the function fail.

REQUEST CONTACT LOOKUP

Result: Normal. No internal error found.

REQUEST CACHE EDIT

Result: Normal. No internal error found.

REQUEST REFRESH CURRENT MAP

Result: Normal. Will timeout if no response is got.

REQUEST COMMIT EDIT

Result: Mostly normal. The extension may hang during the commitment ordering. Will terminate when timeout but the commitment is not made.

REQUEST UNDO COMMIT EDIT

Result: Often cause error. This operation is too expensive and will make the extension hang if the count of affected nodes is large. Also, this part can't run with 100% correctness.

EXIT

Result: Normal. No internal errors found. Every resource is released and the JVM is sure to be terminated.

5.1.2 EXTENSION FUNCTIONALITY

CONTACT LOOKUP

Result: Normal. No internal error found.

ASSEMBLE SUBVERSION

Result: Normal. No internal error found.

ASSEMBLE VERSION

Result: Normal. No internal error found.

COMMIT VERSION

Result: Normal most of the time. The extension may hang during the commitment. When a commitment is committed too fast after it is ordered, the next waiting extension in the queue will have to wait until timeout. This may leads to a sequence of unsuccessful commitments.

PRE-COMMIT VERSION

Result: Normal most of the time. Most version commit errors, especially the ones make the extension hangs, happen in this part.

RELAY PACKET

Result: Normal. But this operation is too expensive. Cause large amount of communications over the network.

VERSION QUERY

Result: Normal. But if an error already exists in the global version id, the result will be wrong.

EXIT

Result: Normal most of the time. But the extension thread may still exist if it is exited while it hangs.

5.2 EVALUATION

5.2.1 OPERATION TIME EFFICIENCY

As observed in the testing, most operations are applied successfully within very short time. Except the version commit operation, which requires waiting in order, all operations are done within 1 second if both users are on the local host.

Most time are spent during the transmission of the packets, as two users on distant hosts spend the same time to finish the work as to simply transfer a message.

5.2.2 OPERATION OVERHEADS

By recording the bandwidth usage when only the extension is running, the overhead of the extension can be calculated. It is not very accurate by this measure but it can be used to show the changes in the amount of overhead.

When testing the overhead generated, each extension only issues version commit commands periodically. Each of the communication takes about the same size of data transmission.

The result is not quite satisfying. The amount of overhead generated is rather random. It is hard to say the overhead is increasing or decreasing, thus, overall, the percent of overhead varies around about 27% during the whole test. This is good enough for the extension to run smoothly.

As no other similar software can be used to compare, no external comparison is made for the overhead generated.

5.2.3 COMMITMENT ORDERING EFFICIENCY

The commit of a version takes an average of about 2 seconds if no error happens when all the extensions are running on the local host. The time needed in a realistic environment should be only this amount added by the average time of transmission.

It can be said that the extension is efficient when no error happens. However, one of the extensions may hang because of the internal bugs. The efficiency of the whole

commit operation is seriously decreased as they need to wait for the timeout (by default 8*5 seconds) to commit.

The commitment ordering in this extension uses as much storage space as a normal commitment ordering. The only additional storage needed is the space used to store the stability of nodes. It is only a double value, so very little additional space is needed.

5.2.4 LOAD-BALANCING EFFICIENCY

The load-balancing only requires a simple calculation that can be done in a few micro-seconds, thus, it hardly affects the efficiency of the extension. Also, the additional space needed is only for storing a few parameters.

5.2.5 VERSION CONTROL EFFICIENCY

It is hard to measure the efficiency of the version control of the extension, as the operations of version control are distributed within the whole working flow.

However, the version control in the extension needs more operations for merging branch versions, the time needed should be increased a little. And the way that versions are assembled by the subversions requires more storage space than the original version control. The efficiency is theoretically decreased by the design. However, all the drawbacks are used gain the ability to easily rollback. The reduced rollback chance and rollback cost makes up for these draw backs.

6 CONCLUSION

An extension for distributed concept mapping is implemented using the improved methods which aim at retaining concurrency correctness in unreliable environment. Most of the functions run correctly with reasonable failure possibility.

The stability parameter guaranteed the correctness and efficiency of the operations in a relatively unreliable environment, though it didn't significantly reduce the overhead of the communications in this extension network.

The commitment ordering with stability method has been observed to be efficient overall. The order of the execution of the commitments is predictable and observed to be correct.

The load-balancing with stability adds little impact on the performance of the extensions, but can successfully balance the communication traffic in the network. Also, instead of blindly distributed the excess work to other nodes, every node will be given work according to their capacity settings.

The modified version control doesn't improve efficiency directly, but it greatly reduced the chance of a rollback, which actually increased the efficiency of the extension even when it fails.

Thus, overall, the extensions can be considered efficient and achieved its objectives.

BIBLIOGRAPHY

[1]Joseph D. Novak, *Learning, Creating, and Using Knowledge: Concept Maps as Facilitative Tools in Schools and Corporations*

[2]Tom Conlon, Steve Gregory, *Representations of Conception: Towards a Repertoire for Thinking and Learning*, 2007
http://www.parlog.com/shared/roc.pdf

[3]Roberto A. Flores-Méndez, *DISTRIBUTED CONCEPT MAPPING COLLABORATION USING JAVA*

[4]Andrew S. Tanenbaum , Maarten van Steen, *Distributed Systems Principle and Paradigms Second Edition*, Prentice Hall , 2006

[5]Yoav Raz, *The Principle of Commitment Ordering, or Guaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Atomic Commitment*, Proceedings of the Eighteenth International Conference on Very Large Data Bases (VLDB), pp. 292-312

[6]Yoav Raz, *Serializability by Commitment Ordering*, Information Processing Letters (IPL), Volume 51, Number 5, pp. 257-264

[7]George F. Coulouris, Jean Dollimore, Tim Kindberg, *Distributed Systems, Concepts and Design Third Edition*, Springer-Verlag New York Inc, 2005

[8]Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley Publishing Company, 1987

[9]Philip A. Bernstein, Nathan Goodman, *Multiversion Concurrency Control – Theory and Algorithms*, ACM Computing Surveys, 1981

[10]Gerhard Weikum, Gottfried Vossen, *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*, Morgan Kaufmann, 2002

[11]David Patrick Reed, *Naming and Synchronization in a Decentralized Computer System*, Massachusetts Institute of Technology Cambridge, MA, USA, 1978

[12]Klaus Haller, Heiko Schuldt, Can Türker, *Decentralized coordination of transactional processes in peer-to-peer environments*, Proceedings of the 2005 ACM CIKM, International Conference on Information and Knowledge Management, pp. 28-35, Bremen, Germany, October 31 - November 5, 2005

[13]Deo P. Vidyarthi,Biplab K. Sarker,Anil Kumar Tripathi,Laurence Tianruo Yang, *Scheduling in Distributed Computing Systems: Analysis, Design and Models*, Springer, 2008

[14]Collins-Sussman, Ben, Fitzpatrick, B.W. and Pilato, C.M., *Version Control with Subversion*. O'Reilly, 2004

[15]Rüdiger Schollmeier, *A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications*, Proceedings of the First International Conference on Peer-to-Peer Computing, IEEE, 2002

[16]John F. Buford, Heather Yu, Eng Keong Lua *P2P Networking and Applications*, Morgan Kaufmann, 2008

[17]Javed I. Khan and Adam Wierzbicki, *Foundation of Peer-to-Peer Computing, Special Issue, Elsevier Journal of Computer Communication*, (Ed) Volume 31, Issue 2, 2008

[18]Tel G, *Introduction to Distributed Algorithms*. Cambridge, UK: Cambridge University Press,

2nd ed., 2000
[19]Tanisch P, *Atomic Commit in Concurrent Computing*, IEEE Concurrency, (8)4:34-41, 2000
[20]Wiesmann M., Pedone F., Schipher A., Kemme, B, Alonso G, *Understanding Replication in Databases and Distributed Systems*, 2000
[21]Sun, *Java SE 6.0 API Specification*
http://download.oracle.com/javase/6/docs/api/
[22]Daudjee, K.,Salem, K., *Lazy database replication with ordering guarantees*, Data Engineering, 2004.Proceedings. 20th International Conference on 2004 pages: 424 - 435, 2004
[23]Raz, Y., *Commitment ordering based distributed concurrency control for bridging single and multi version resources*, Third International Workshop on 1993 pages: 189 - 198, 1993
[24]Daudjee, K., Salem, K., *Inferring a Serialization Order for Distributed Transactions*, Proceedings of the 22nd International Conference on 2006 pages: 154 - 154, 2006
[25]Martin, R.,Menth, M.,Hemmkeppler, M., *Accuracy and Dynamics of Hash-Based Load Balancing Algorithms for Multipath Internet Routing*, 3rd International Conference on 2006 pages: 1 - 10, 2006
[26]Ka-Po Chow,Yu-Kwong Kwok, *On load balancing for distributed multiagent computing*, Parallel and Distributed Systems, IEEE Transactions on 2002 volume: 13 issue: 8 pages: 787 - 801, 2002
[27]Godfrey, B. et al., *Load balancing in dynamic structured P2P systems*, Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies 2004 volume: 4 pages: 2253 - 2262 vol.4, 2004
[28]Kathuria, Vishal, *Transaction isolation and lazy commit*, Proceedings - International Conference on Data Engineering VOL ISSU PAGE 1204-1211 DATE 2007, 2007
[29]Salzberg, B., *Timestamping after commit*, Proceedings of the Third International Conference on 1994 pages: 160 - 167, 1994
[30]Bieniusa, Annette, *The relation of version control to concurrent programming*, Proceedings - International Conference on Computer Science and Software Engineering, CSSE 2008 VOL 3 ISSU PAGE 461-464 DATE 2008, 2008
[31]Mukherjee, Patrick, *Peer-to-peer based version control*, Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS VOL ISSU PAGE 829-834 DATE 2008, 2008
[32]Hinsen, Konrad, *Essential tools: Version control systems*, Computing in Science and Engineering VOL 11 ISSU 6 PAGE 84-91 DATE November-December 2009, 2009
[33]Cay S. Horstmann, Gary Cornell.,*Core Java*, 2008
[34]Goetz, Brian., *Java concurrency in practice*, 2006
[35]Jeff Magee & Jeff Kramer., Concurrency : state models & Java programs, 2006
[36]Motta, Rossana, Gnutella: Integrating performance and security in fully decentralized P2P models, Proceedings of the 46th Annual Southeast Regional Conference on XX, ACM-SE 46 VOL ISSU PAGE 272-277 DATE 2008, 2008
[37]Dinger, Jochen, Decentralized bootstrapping of P2P systems: A practical view, Lecture Notes in Computer Science VOL 5550 LNCS ISSU PAGE 703-715 DATE 2009, 2009
[38]Bocek, Thomas, Peer vote: A decentralized voting mechanism for P2P collaboration systems, Lecture Notes in Computer Science VOL 5637 LNCS ISSU PAGE 56-69 DATE 2009, 2009

Appendices

There is a lot of coding in this project, thus the five pages is not enough for all the essential part of extension.

Send a packet

```
public static boolean sendPacket(Packet packet,int timeout){
        Socket socket = null;
        try {
            socket = new Socket(CommonServices.getHost(packet.getTargetId()),
CommonServices.getPort(packet.getTargetId()));
            if(timeout>0)
                socket.setSoTimeout(timeout);
            socket.setSoLinger(false, 0);
            ObjectOutputStream packetStream = new ObjectOutputStream(socket.getOutputStream());
            packetStream.writeObject(packet);
            socket.close();
        } catch (ConnectException e) {
            //Starter.println("<[Connect]Failure>"+e.getMessage());
            return false;
        } catch (SocketException e) {
            //Starter.println("<[Socket]Failure>"+e.getMessage());
            return false;
        } catch (UnknownHostException e) {
            //Starter.println("<[UnknownHost]Failure>"+e.getMessage());
            return false;
        } catch (IOException e) {
            //Starter.println("<[IO]Failure>"+e.getMessage());
            return false;
        }
        return true;
    }
```

Version Commit

```
String mapId = (String)this.para[0];
//notify local version control and collect current version.
version = this.starter.versionControl.preCommitVersion(mapId);
if(version!=null){
    //lock commit status
    this.starter.versionControl.lockCommitStatus(mapId);
    //wait for local version control to assign id to this version
    String[] rcContactList;
    rcContactList = this.starter.contacts.getContactList(mapId);
    if(rcContactList==null){
        rcContactList = this.starter.contacts.getContactsList();
    }
    synchronized(this.version){
        //wait until an id is allocated
        while((this.version.getId()==-1)){
            try {
```

```
                    this.version.wait(3*1000);
                } catch (InterruptedException e) {
                    this.starter.userOutput("<Interrupted>");
                    break;
                }
            }
        }
        if(this.version.getId()!=-2){
            //update local version history;
            this.starter.versionHistory.newVersion(this.version);
            String next = this.starter.versionControl.getNextNotify(mapId);
            boolean nextNotified = false;
            //request remote version update
            for(int i = 0;i<rcContactList.length;i++){
                if(rcContactList[i] == next)
                    nextNotified = true;
                this.starter.execute(new DispatcherVersCommitCmd(this.starter,rcContactList[i],this.version));
            }
            if((!nextNotified)&&(next!=null)){
                this.starter.execute(new DispatcherVersCommitCmd(this.starter,next,this.version));
            }
            //unlock commit status
            this.starter.versionControl.unlockCommitStatus(mapId);
            this.starter.userOutput("new version committed:"+this.version.getId());
        }
        else{
            this.starter.userOutput("version commit timeout.");
        }
}
else{
        this.starter.userOutput("No new version to commit.");
}
```

Version Pre-commit

```
//broadcast pre-requests
String[] idList = this.starter.contacts.getContactsList();
this.voteAll = idList.length;
//for testing
long delay = (Math.round(Math.random()*3)+2);
this.starter.userOutput("delay :"+delay+" seconds");
try {
        Thread.sleep(delay*1000);
} catch (InterruptedException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
}
//send out pre-commit(vote) requests
for(int i = 0;i<idList.length;i++){
        this.starter.execute(new DispatcherVersPreCommitCmd(this.starter,idList[i],this));
}
```

```
//wait for votes,timeout = 8*5 seconds.
int i = 0;
while((!this.isChecked())&&(i<7)){
	i++;
	synchronized(this.approvedLock){
		try {
			this.approvedLock.wait(5*1000);
		} catch (InterruptedException e) {
			this.starter.userOutput("<Interrupted>");
			break;
		}
	}
}
if(i>=7){
	//timeout. notify the user.
	this.starter.userOutput("<[Timeout]Failure>VersionPreCommit for "+this.mapId+" timeout.");
	this.version.setId(-2);
}
else{
	//commit condition
	this.approvedLock.lock();
	if(this.approved){
		this.approvedLock.unlock();
		this.starter.userOutput("All votes approve,commit");
	}
	else{//must have negative vote;
		this.starter.userOutput("Has negative vote");
		this.approvedLock.unlock();
		//wait for commit
		while(!this.starter.versionControl.isEmpty(this.mapId)){
			synchronized(this){
				try {
					this.wait();
				} catch (InterruptedException e) {
					this.starter.userOutput("<Interrupted>");
					break;
				}
			}
		}
		this.starter.userOutput("Reached head of the queue,commit");
	}
	//set new version id
	int k =this.starter.versionHistory.getVersionId(this.mapId);
	this.version.setId(k+1);
}
//notify version commit
synchronized(this.version){
	this.version.notify();
}
```

Command Listener

```
public void run() {
	try {
		this.commandStream = new ObjectInputStream(this.starter.socket.getInputStream());
	} catch (IOException e) {
		this.starter.userOutput("<[IO]Failure>"+e.getMessage());
		e.printStackTrace();
	}
	while(!this.starter.exit){
		try {
			int commandId = this.commandStream.readInt();
			Object[] commandPara = null;
			if(this.commandStream.readBoolean())
				try {
					commandPara = (Object[]) this.commandStream.readObject();
				} catch (ClassNotFoundException e) {
					this.starter.userOutput("<[ClassNotFound]Failure>Command Structure Invalid! "+e.getMessage());
					continue;
				}
			CommandsUser command = CommandsUser.getConceptionCommand(commandId);
			if(command!=null){
				command.execute(this.starter,commandPara);
			}
			else{
				this.starter.userOutput("Invalid command id.");
			}
		} catch (IOException e) {
			this.starter.userOutput("<[IO]Failure>"+e.getMessage());
			e.printStackTrace();
		}
	}
}
```

Contacts List

```
	private HashMap<String,ContactDetail> primaryContacts;
	private LinkedList<CustomEntry<String,Double>> WeightedContacts;
	private HashMap<String,ContactDetail> secondaryContacts;
	private HashMap<String,Set<String>> rcList;

	private boolean contains(String id){
		contactsLock.lock();
		if(primaryContacts.containsKey(id)){
			contactsLock.unlock();
			return true;
		}
		contactsLock.unlock();
		sContactsLock.lock();
		if(secondaryContacts.containsKey(id)){
			sContactsLock.unlock();
			return true;
```

```
        }
        sContactsLock.unlock();
        return false;
    }
    private boolean contains(String id,boolean isResponce){
        contactsLock.lock();
        if(primaryContacts.containsKey(id)){
            if(isResponce)
                primaryContacts.get(id).respReceived++;
            contactsLock.unlock();
            return true;
        }
        contactsLock.unlock();
        sContactsLock.lock();
        if(secondaryContacts.containsKey(id)){
            if(isResponce)
                secondaryContacts.get(id).respReceived++;
            sContactsLock.unlock();
            return true;
        }
        sContactsLock.unlock();
        return false;
    }
    private ContactDetail get(String id){
        ContactDetail contact;
        contactsLock.lock();
        contact = primaryContacts.get(id);
        if(contact!=null){
            contactsLock.unlock();
            return contact;
        }
        contactsLock.unlock();
        sContactsLock.lock();
        contact = secondaryContacts.get(id);
        if(contact!=null){
            sContactsLock.unlock();
            return contact;
        }
        sContactsLock.unlock();
        return null;
    }
    public boolean put(String id,ContactDetail contact){
        if(!(contactsCount<this.starter.config.getPrimaryContactsThreshold())){
            if((contact.avgRespTime>maxRespTime)||(contact.stability<minStability)){
                if(!(contactsCount>=this.starter.config.getContactsLimit()*2)){
                    sContactsLock.lock();
                    secondaryContacts.put(id, contact);
                    sContactsLock.unlock();
                }
```

```
                return false;
            }
        }
        else{
            maxRespTime = (maxRespTime<contact.avgRespTime)?contact.avgRespTime:maxRespTime;
            minStability = (minStability>contact.stability)?contact.stability:minStability;
        }
        contactsLock.lock();
        primaryContacts.put(id, contact);
        contactsLock.unlock();
        double weight = calcWeight(contact);
        WeightedContacts.add(searchIndexOf(weight), new CustomEntry<String,Double>(id,weight));
        return true;
    }
    private void move(String id,boolean isPrimary){
        if(isPrimary){
            contactsLock.lock();
            ContactDetail contact = primaryContacts.get(id);
            if(contact!=null){
                primaryContacts.remove(id);
                contactsLock.unlock();
                sContactsLock.lock();
                secondaryContacts.put(id, contact);
                sContactsLock.unlock();
            }
            else
                contactsLock.unlock();
        }
        else{
            sContactsLock.lock();
            ContactDetail contact = secondaryContacts.get(id);
            if(contact!=null){
                secondaryContacts.remove(id);
                sContactsLock.unlock();
                contactsLock.lock();
                primaryContacts.put(id, contact);
                contactsLock.unlock();
            }
            else
                sContactsLock.unlock();
        }
    }
    private void remove(String id){
        sContactsLock.lock();
        secondaryContacts.remove(id);
        sContactsLock.unlock();
    }
    private int searchIndexOf(double weight){
        if(WeightedContacts.size()==0){
```

```
                return 0;
        }
        int i,j;
        for(i=0,j=WeightedContacts.size();i<j;){
                double temp = WeightedContacts.get((i+j)/2).getValue();
                if(temp==weight)
                        return 1+(i+j)/2;
                else if(temp>weight)
                        j=(i+j)/2;
                else
                        i=(i+j)/2;
        }
        return 1+i;
}
public boolean addContact(String id, boolean isResponce){
        if(id.equals(this.starter.getMyId()))
                return false;
        ContactDetail contact = new ContactDetail();
        if(isResponce)
                contact.respReceived++;
        if(contains(id,isResponce)){
                return false;
        }
        this.starter.contacts.put(id,contact);
        contactsCount++;
        return true;
}
public String[] getContactsList(){
        contactsLock.lock();
        Set<String> contactsSet = primaryContacts.keySet();
        String[] rtv = new String[contactsSet.size()];
        Iterator<String> itr = contactsSet.iterator();
        int i = 0;
        while(itr.hasNext()&&i<rtv.length){
                rtv[i] = itr.next();
                i++;
        }
        contactsLock.unlock();
        return rtv;
}
public boolean setActivePrim(String id,boolean active){
        ContactDetail contact;
        contactsLock.lock();
        contact = primaryContacts.get(id);
        if(contact!=null){
                contact.active = active;
                contactsLock.unlock();
                if(active = false)
                        move(id,true);
```

```
                return true;
        }
        contactsLock.unlock();
        sContactsLock.lock();
        contact = secondaryContacts.get(id);
        if(contact!=null){
                contact.active = active;
                sContactsLock.unlock();
                return true;
        }
        sContactsLock.unlock();
        return false;
}
public double getStability(String id){
        ContactDetail contact;
        double rtv = -1;
        contactsLock.lock();
        contact = primaryContacts.get(id);
        if(contact!=null){
                rtv = contact.stability;
                contactsLock.unlock();
                return rtv;
        }
        contactsLock.unlock();
        sContactsLock.lock();
        contact = secondaryContacts.get(id);
        if(contact!=null){
                rtv = contact.stability;
                sContactsLock.unlock();
                return rtv;
        }
        sContactsLock.unlock();
        return rtv;
}
public boolean updateStability(String id) {
        ContactDetail contact;
        contactsLock.lock();
        contact = primaryContacts.get(id);
        if(contact!=null){
                double nStability = (double)contact.respReceived/(double)contact.reqSent;
                double runTime = this.starter.runTime();
                nStability += ((double)System.currentTimeMillis() - (double)contact.lastUpTime)/runTime;
                contact.stability = contact.stabilityFactor*Math.sqrt(nStability);
                contact.stabilityFactor = Math.sqrt(contact.stability/2);
                contactsLock.unlock();
                return true;
        }
        contactsLock.unlock();
        sContactsLock.lock();
```

```
        contact = secondaryContacts.get(id);
        if(contact!=null){
            double nStability = (double)contact.respReceived/(double)contact.reqSent;
            double runTime = this.starter.runTime();
            nStability += ((double)System.currentTimeMillis() - (double)contact.lastUpTime)/runTime;
            contact.stability = contact.stabilityFactor*Math.sqrt(nStability);
            contact.stabilityFactor = Math.sqrt(contact.stability/2);
            sContactsLock.unlock();
            return true;
        }
        sContactsLock.unlock();
        return false;
    }
    public boolean reqSent(String id){
        ContactDetail contact;
        contactsLock.lock();
        contact = primaryContacts.get(id);
        if(contact!=null){
            contact.reqSent++;
            contactsLock.unlock();
            return true;
        }
        contactsLock.unlock();
        sContactsLock.lock();
        contact = secondaryContacts.get(id);
        if(contact!=null){
            contact.reqSent++;
            sContactsLock.unlock();
            return true;
        }
        sContactsLock.unlock();
        return false;
    }
    public boolean respReceived(String id,long respTime){
        ContactDetail contact;
        contactsLock.lock();
        contact = primaryContacts.get(id);
        if(contact!=null){
            contact.avgRespTime = (contact.avgRespTime*contact.respReceived+respTime)/(contact.respReceived++);
            contactsLock.unlock();
            return true;
        }
        contactsLock.unlock();
        sContactsLock.lock();
        contact = secondaryContacts.get(id);
        if(contact!=null){
            contact.avgRespTime = (contact.avgRespTime*contact.respReceived+respTime)/(contact.respReceived++);
            sContactsLock.unlock();
            return true;
```

```
        }
        sContactsLock.unlock();
        return false;
    }
    public double calcWeight(ContactDetail contact){
        double rtv = 0;

        return rtv;
    }
    public String[] getRcList(){
        rcLock.lock();
        Set<String> rcSet = rcList.keySet();
        String[] rtv = new String[rcSet.size()];
        Iterator<String> itr = rcSet.iterator();
        int i = 0;
        while(itr.hasNext()&&i<rtv.length){
            rtv[i] = itr.next();
            i++;
        }
        rcLock.unlock();
        return rtv;
    }
    public String[] getContactList(String mapId){
        rcLock.lock();
        Set<String> rcSet = rcList.get(mapId);
        if(rcSet == null){
            rcLock.unlock();
            return null;
        }
        String[] rtv = new String[rcSet.size()];
        Iterator<String> itr = rcSet.iterator();
        int i = 0;
        while(itr.hasNext()&&i<rtv.length){
            rtv[i] = itr.next();
            i++;
        }
        rcLock.unlock();
        return rtv;
    }
```