eQuus Protocol Specification

Jun Li

19th February 2007

Tutor: Thomas Locher, Stefan Schmid, Roger Wattenhofer

Abstract

eQuus Protocol (QP) makes use of TCP/IP network infrastructures in order to provide a means of decentralized communication based on logic ID routing, and of organizing a large number of dynamic peers in the network when joining and leaving are frequent. The specification defines services, interfaces and protocols of QP.

Contents

1 Introduction 4

2 Overview 5
2.1 Scope . . . 5
2.2 Purpose . . . 6

3 Definitions and Abbreviations 7
3.1 Definitions . . . 7
3.1.1 Peer . . . 7
3.1.2 Clique . . . 7
3.1.3 Router . . . 7
3.1.4 Supplicant . . . 7
3.1.5 Member . . . 8
3.1.6 Coordinator . . . 8
3.1.7 Participant . . . 8
3.1.8 DHT . . . 8
3.2 Abbreviations . . . 8

4 Conformance 8

5 Principles of Operation 9
5.1 Peers, Cliques, and Roles . . . 9
5.2 Route . . . 11
5.3 Lookup . . . 11
5.4 Routing Table Update . . . 12
5.5 Measure Distance . . . 13
5.6 Join and Leave . . . 13
5.7 Establish Consistency . . . 14
5.8 Merge and Split . . . 14
5.9 DHT . . . 16

6 QP Encapsulation over TCP/IP 16
6.1 Transmission and Representation of Octets . . . 17
6.2 QP Packet Format for TCP . . . 17
6.3 QP Packet Format for UDP . . . 17
6.4 QP PDU Field and Parameter Definitions . . . 17
6.4.1 Protocol Identifier . . . 17
6.4.2 Protocol Version . . . 18
6.4.3 Packet Type . . . 18
6.4.4 Packet Body Length . . . 20

6.4.5 Packet Body 20

A View-Synchronous Reliable Broadcast **28**

B Augmented BNF **30**

1 Introduction

This specification defines a mechanism for peer-to-peer communication that makes use of current TCP/IP network infrastructures to organize peers and their communication based on logic ID routing in the dynamic Internet-scale environment. The relation of the specification and other network protocols is shown below in Figure1.

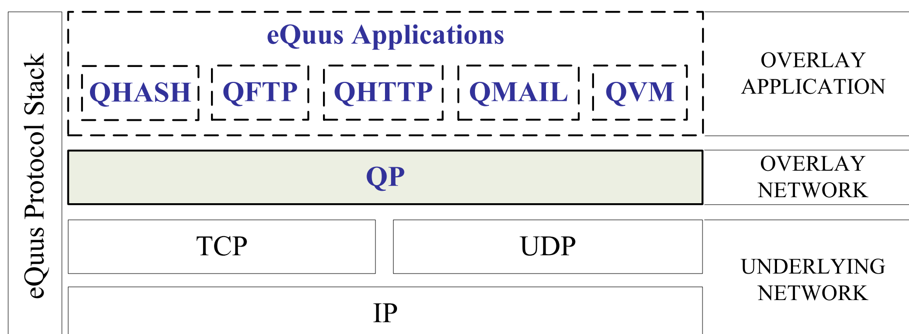

Figure 1: From bottom to up, eQuus protocol stack logically consists of five layers. The two lowest layers are IP and TCP/UDP which construct the current underlying network. Built upon them, eQuus organizes logic peers with the mapping of physical machines into an overlay network in which application layer protocols and peer-to-peer applications use QP to communicate with each other in a decentralized way.

The core of the protocol stack is QP which supports the communication of peer-to-peer applications. QP can use either TCP or UDP[1] for peer communication. Several application layer protocols, such as QHASH(eQuus Hash Function) - a hash function mechanism to map entities to logic IDs, QFTP(eQuus File Transfer Protocol) - a protocol for file sharing applications, QHTTP(eQuus Hypertext Transfer Protocol) - for transferring HTTP messages based on logic IDs, QMAIL(eQuus Mail Protocol) - for sending and receiving emails and QVM(eQuus Virtual Machine) - a protocol that constructs virtual computers based on a group of specified peers[2], are built upon QP to support specific types of applications.

[1]The choice of where and when using TCP or UDP protocol is up to communication type. For communication among the peers in the same clique, QP uses TCP. For communication among the peers in different cliques, QP uses UDP for the sake of reducing cost. In addition, the upper layers of eQuus protocol stack are logically dependent on QP for resolving ID lookup and routing messages. Applications usually define their own protocols and use TCP/UDP to communicate target peers after resolving their underlying addresses. In case of routing anomalies, asymmetric links and communication behind NATs/firewalls, the upper layers try to send and receive messages through QP.

[2]Comparing to traditional application layer protocols, some QP based protocols have the similar functionalities but present different interaction modes. Others, such as QHASH and QVM, are introduced to facilitate new generation applications. A distinguished difference between eQuus and traditional network is the protocol of QHASH that maps all kinds of resources into logic IDs and assigns them to corresponding logic peers.

The ultimate goal of eQuus system is to construct a peer-to-peer overlay network and applications without centralized servers. This specification focuses on QP which builds and maintains the eQuus overlay.

2 Overview

2.1 Scope

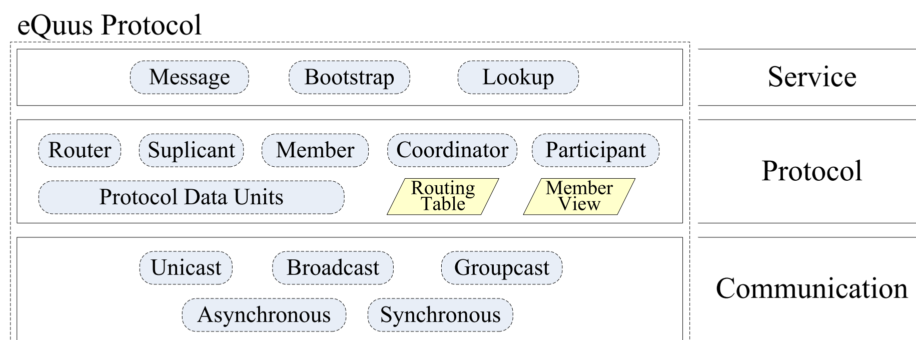

Figure 2: Layers in eQuus protocol consists of three parts. Service layer is for eQuus applications which send/receive messages, join eQuus system and lookup specific IDs. Protocol layer defines different entities and their behaviors to implement the provided services and to maintain eQuus network. It also specifies PDUs, routing table and member view. Communication layer defines different types of communication facilities.

QP is deployed on computers that need peer-to-peer interaction facility. It enables high-level protocols, services or applications communicate with the corresponding parts residing on other computers by logic IDs. Two primary functions of QP are peer-to-peer interaction primitives and autonomous peer management as showed in Figure2. QP provides interaction primitives in the form of services, including message service[3] which means that given an ID, QP forwards messages hop by hop until reaching the destination peer who is responsible for the ID, bootstrap which means that controls the host peer to be the member of eQuus overlay network, lookup for finding several peers in a clique according to its ID. Autonomous peer management includes clique organization and membership management. Every eQuus clique can handle join request from any individual peer and merge or split clique when the number of members in some clique is too little or too much. Clique organization makes eQuus overlay network fully-decentralized and self-organized. Membership management ensures

[3]The main purpose of message service is for transmitting control messages between two peers based on IDs and not directly for data transmission. It supports both synchronous and asynchronous communication. Unlike TCP/UDP that differentiates potential receivers with various ports, message service chooses receivers based on message contents.

all members in a clique have a consistent view of their membership. Both functions are built upon communication facilities in QP. Instead of specifying data transmission mechanisms such as end-to-end data reliability, flow control and sequencing, which the lower TCP/UDP level has already addressed, QP defines high level communication facilities including unicast, broadcast or groupcast[4] and synchronous/asynchronous communication by wrapping TCP/UDP primitives.

2.2 Purpose

For the purpose of providing compatible interaction and peer management mechanisms for peers connected through eQuus protocols, this specification specifies a general method for the provision of eQuus Protocol. To this end, it

- Describes the architectural framework within which the interaction and peer organization take place
- Defines the principles of operation of eQuus interaction and management mechanisms
- Defines the different levels and types of communication that are supported and the behavior of the peer with respect to the transmission and reception of messages when performing peer interaction or management
- Establishes the requirements for a protocol between the peer that requires message route to take place and the peer that is responsible for the destination ID and for a protocol among peers in the middle hops
- Establishes the requirements for a protocol to perform lookup operation
- Establishes the requirements for a protocol to update routing table
- Establishes the requirements for a protocol to measure peer distance
- Establishes the requirements for a protocol to join and leave eQuus network
- Establishes the requirements for a protocol among all the members in a clique to keep membership consistency
- Establishes the requirements for protocols to address peer merge and split in cliques

[4]The different between broadcast and groupcast is that broadcast does not require consistency while groupcast is a kind of communication among a group of peers in the same clique which must have a consistent view of their membership. Groupcast probably uses view-synchronous broadcast algorithm.

- Establishes the requirements for a protocol to implement distributed hashtable mechanism in eQuus network
- Specifies mechanisms and procedures that support eQuus interaction and management through the use of eQuus Protocol
- Specifies the encoding of the Protocol Data Units (PDUs) used in interaction and management protocol exchanges
- Specifies the requirements to be satisfied by equipment claiming conformance to this specification

3 Definitions and Abbreviations

For the purposes of this specification, the following terms, definitions, acronyms, and abbreviations apply.

3.1 Definitions

3.1.1 Peer

An entity at one end point of the underlying network that is also the entity attached to the eQuus overlay network. Each peer in the eQuus network has a unique ID which is a bit string of a predefined length d.

3.1.2 Clique

A group of peers that are close-by. Within such a clique, each peer has the same ID.

3.1.3 Router

A protocol entity performing routing messages and resolving IDs. When receiving a message, router will forward it according to its routing tables. If the destination is the host peer, it will deliver the message to upper layers. It can also resolve ID to a list of members in the corresponding clique.

3.1.4 Supplicant

A protocol entity running on a peer that actively controls the peer to join or leave the eQuus network[5].

[5] Since the peer hasn't joined the eQuus network, supplicant can't use communication facilities in eQuus to communicate with other peers. In such case, it must use some mechanisms such as broadcast in IP layer or peer cache to cooperate with some members to initialize its routing tables.

3.1.5 Member

A protocol entity running on a peer that passively receives the request of a new peer to join or leave the eQuus network and helps the new peer to be or not to be part of the the eQuus network[6] It is also responsible for distance measurement and keeping membership consistency.

3.1.6 Coordinator

A protocol entity playing the role of commander during the course of clique merge and split.

3.1.7 Participant

A protocol entity playing the role of participants during the course of clique merge and split.

3.1.8 DHT

A protocol entity playing the role of maintaining a distributed hashtable for all peers in the system and serving requests like *get* or *put* in normal hashtable.

3.2 Abbreviations

QP	eQuus Protocol
QHASH	eQuus Hash Function
QFTP	eQuus File Transfer Protocol
QHTTP	eQuus File Transfer Protocol
QFTP	eQuus Hypertext Transfer Protocol
QMAIL	eQuus Mail Protocol
QVM	eQuus Virtual Machine
DHT	Distributed Hash Table

4 Conformance

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL”in this document are to be interpreted as as described in RFC 2119 [2].

[6] When happening peer failure, the members in the same clique should set a threshold to determine the leave of the failure peer by periodically checking its liveness.

An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements for the protocols it implements. An implementation that satisfies all the MUST or REQUIRED level and all the SHOULD level requirements for its protocols is said to be "unconditionally compliant"; one that satisfies all the MUST level requirements but not all the SHOULD level requirements for its protocols is said to be "conditionally compliant".

5 Principles of Operation

This clause describes the architectural framework of eQuus Protocol and the relationship between the interaction and organization function and the operation of the peers within which it is deployed. The detailed algorithms and proofs are referred to [5].

5.1 Peers, Cliques, and Roles

Computers in underlying network that attach to eQuus network, referred to this specification as Peers (3.1.1), are partitioned into different groups, referred to this specification as Cliques (3.1.2). In eQuus network, peers and cliques can perform different kinds of operations to support peer interaction and network maintenance. Peers can route messages to the destination, perform lookup based on logic IDs and keep the correctness of routing table. New peers can join eQuus network and old peers can leave it without any centralized control. All members of a clique keep the same view during the lifetime of the clique. Cliques can merge or split according to the number and thresholds of their members. Peer can associate data with keys or detach them in a distributed way. For the purposes of describing the operation of eQuus interaction, a peer is able to adopt distinct roles within an interaction of some operation.

For routing messages, a peer should adopt the following role:

- Router (3.1.3): The peer that takes the responsibility of forwarding messages to destination adopts the Router role.

For performing lookup, a peer should adopt the following role:

- Router (3.1.3): The peer that takes the responsibility of requesting and replying lookup messages adopts the Router role.

For updating routing table to maintain correctness, a peer should adopt the following role:

- Router (3.1.3): The peer that takes the responsibility of handling updating routing table messages and of keeping the correctness of routing table adopts the Router role.

For measuring peer distance, a peer should adopt the following role:

- Member (3.1.5): The peer that takes charge of measuring peer distance, including sending and replying measurement messages, adopts the Member role.

For joining and leaving eQuus network, a peer should adopt the following two roles:

- Supplicant (3.1.4): The peer that wishes to join or leave eQuus network adopts the Supplicant role;
- Member (3.1.5): The peer that takes charge of maintaining the membership of a clique, including peer join/leave and membership consistency, adopts the Member role.

For establishing membership consistency, a peer should adopt the following role:

- Member (3.1.5): The peer that takes charge of notifying membership change and of updating membership view, adopts the Member role.

For merging and splitting cliques, a peer in some clique should adopt the following two roles:

- Coordinator (3.1.6): The peer in a clique that is responsible for controlling the whole process of clique merging and splitting adopts the Coordinator role;
- Participant (3.1.7): The peer in a clique that participants the process of clique merging and splitting and votes its coordinator if there is no coordinator in current clique adopts the Participant role.

For providing DHT service, a peer should adopt the following role:

- DHT (3.1.8): The peer that takes charge of constructing and maintaining a distributed hashtable, and serving requests like normal hashtable, adopts the DHT role.

As can be seen from these descriptions, all these roles are necessary to construct, maintain, and communicate through eQuus network. In a typical eQuus network environment, each peer in the system should be capable of adopting all the roles above since every peer is identical to others and is able to perform all operations. Although eQuus is an open peer-to-peer environment, in reality member role may utilize some authentication mechanism to restrict peer join.

5.2 Route

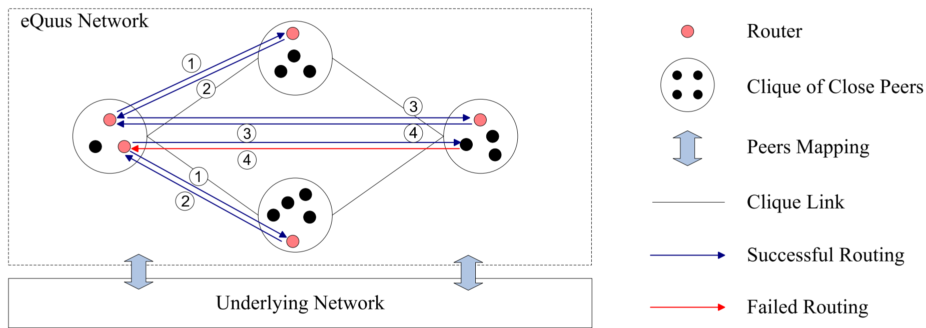

Figure 3: Iterative Routing in eQuus Network

Figure3 illustrates two cases of the route operation in eQuus network. If a clique links to another clique, each peer in the source clique connects different k peers of the target clique. Route operation is initiated from some peer in a clique and is performed hop by hop iteratively through peer connection based on eQuus routing algorithm. The final destination peer is the one that takes charge of the destination ID. If there is no such peer holding the destination ID, the route operation is failed and a failed message is reported to the source peer.

Router role resides along the path from the source peer to the destination peer. Router in source peer accepts messages from its upper layers and contacts routers iteratively till finding correct destination router or failed. When router in destination peer receives the messages, it delivers them to its upper layers. Routers in the middle hops are only responsible for replying next hop information.

5.3 Lookup

Figure4 illustrates lookup operation between two peers. It is initiated by some source peer who wants to know the list of peers in a clique that is responsible for the specific ID. The destination peer replies the request with a list of network addresses of active peers in the same clique. Request and response messages are

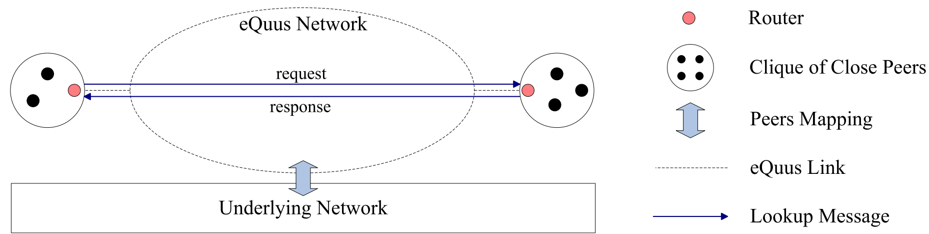

Figure 4: Lookup between Source and Destination

implemented in route messages. Router role residing in the source peer takes charge of performing lookup operation.

5.4 Routing Table Update

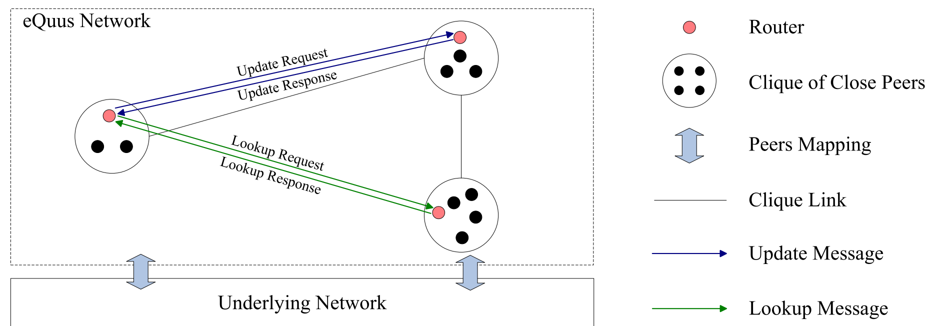

Figure 5: Routing Table Update Operation

The operation of routing table update is triggered periodically to refresh routing table. During the operation, the update peer should deal with two kinds messages, update message and lookup message. For those entries which are filled with specific clique information, the update peer sends update message to a group of addresses in that clique until one of them replies a list of refresh peers. If all the peers in that clique are unreachable, the update peer removes the entry. For those entries which are empty in the original routing table, the update peer performs lookup operation to find responding cliques. If some returned clique happens to be a correct entry, the routing table will be updated. Otherwise, the entry keeps empty. Router role residing in the initiating and corresponding peer takes charge of performing sending update messages and lookup messages.

5.5 Measure Distance

Distance measurement is important in eQuus system since one property of eQuus is to achieve locality and small stretch. Measurement operation is used to organize members belonging to a clique. When a new peer wants to join some clique, it should choose the nearest one. Also, when a clique is splitted into two parts, the arrangement of peers should be determined by their physical location with respect to inter/intra clique distance measurement.

5.6 Join and Leave

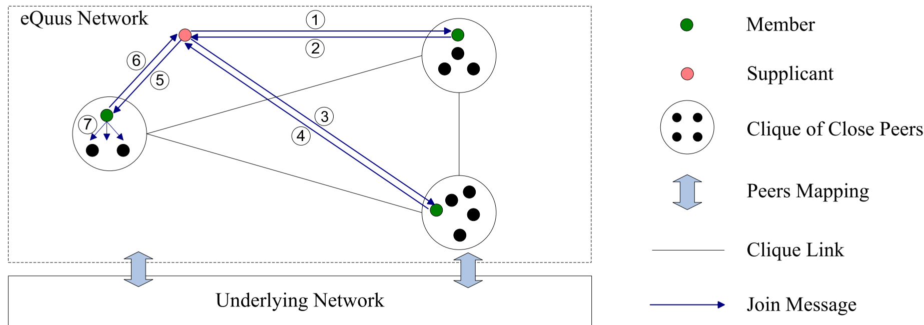

Figure 6: Join Operation for Supplicant

During the lifetime of eQuus network, new peers may join in some cliques and old peers may leave network without notice. Join operation ensures that all peers belonging to the same clique are close to each other. A newly arriving peer must join the closest clique in the system. To achieve it, the new peer as a supplicant first contacts an arbitrary bootstrap member peer. The contacted member peer returns the address of one peer of each clique in its routing table. This process is repeated until the closest clique[7] has been found or an upper bound[8] on the number of rounds has been exceeded. After the new peer is accepted by a member in the clique, the member broadcasts the information of the new peer to all other members. A typical join process is showed in Figure6. The sequence number indicates the order of messages exchanged between the supplicant and members.

Note that there is no need for a specific leave operation. After a peer could not be contacted by any clique member for a certain period of time, it is simply excluded from the clique and thus, from the system. Any change of the membership of a clique, including peer joining or leaving, would cause membership

[7]The closest clique is measured by Round Trip Time (RTT) between supplicant and member. It is usually calculated from distance measurement operation.

[8]The upper bound is used to avoid looping between cliques.

consistency operation to take effect.

5.7 Establish Consistency

To keep eQuus network operate correctly, all the members in a clique should have a consistent view of their membership. Every peer as a role of member receives join request and runs a failure detector to discover faulty peers. It outputs a sequence of group membership sets that are called views. Every view is delivered through view flush message in which a view id denotes a monotonically increasing identifier.

Two kinds of events would cause members in a clique to reestablish membership consistency. Whenever a member accepts a join request from some newly arriving peer, it broadcasts a view flush message to all other clique members. It is also the same action with the case when the failure detector of a member detects some failed members. Member role can be implemented using an eventually perfect failure detector and requires a timing assumption.

Membership consistency is often combined with reliable (FIFO, causally ordered, or atomic) broadcast to implement view-synchronous group communication. In the communication part of QP, groupcast facility is introduced to provide view-synchronous broadcast to a single clique. A distinguished characteristic of groupcast is that peer who receives a groupcast message only delivers to the upper layer if the source peer has the same membership view id of the destination peer. Meanwhile, the applications are blocked during the transition of membership view. The detailed of view-synchronous group communication is referred in in Appendix A.

5.8 Merge and Split

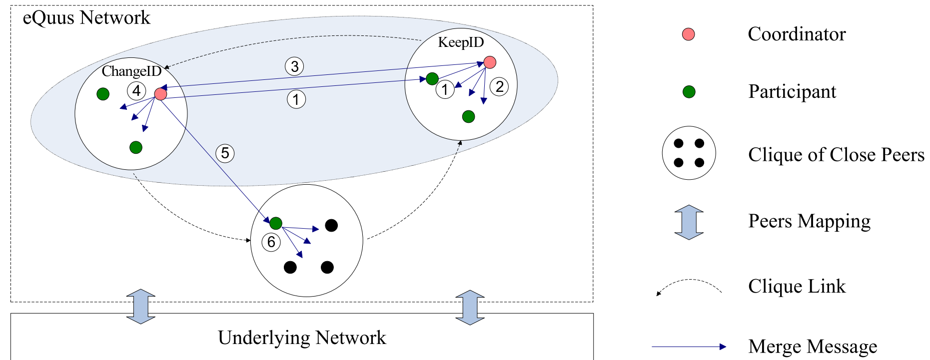

Figure 7: Two Cliques Merge Together

Merge operation occurs when the number of members in some clique is too

small. It is always initiated by the coordinator in the clique without enough members to merge with its predecessor clique. As Figure7 shows, the coordinator in the request clique first picks a peer as a participant in its predecessor and sends a merge request message. When the participant receives it, it resends the request to the coordinator in its clique. After that, the coordinator forwards new state information to the other clique members which update their old state information accordingly. Then, it replies merge request with current state information after refreshing itself. When the coordinator in the request clique receives the state information of its predecessor clique which is needed to update its tables, it broadcasts to all clique members. The successor clique is also informed about merging in order to adapt its predecessor link.

Before some peer conduct an operation in its clique, it should establish membership consistency first. e.g. when a coordinator sends the merge request or a participant broadcasts new state information to its clique members, they both need consistent state. In addition, merge operations must be serialized. It means that if there are two merge operations whose merge clique sets intersect, one must follow the other.

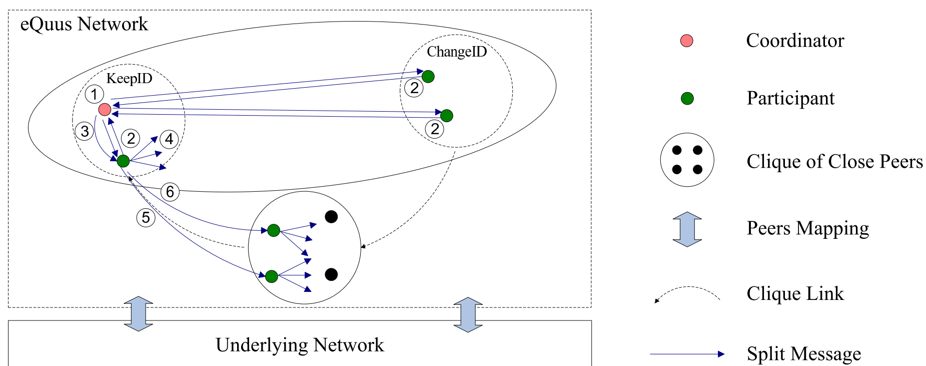

Figure 8: Clique Split

In case the size of a clique exceeds the upper bound, split operation is performed in which half of the peers form a new clique. The peers in the old and the newly created clique are then responsible for approximately half of the IDs they were responsible before. In order to ensure a certain locality, the peers closer to the preceding clique in the ID space keep their ID while the others get the new, higher ID, which is the ID between the current ID and the ID of the successor clique. The closest node to the preceding clique determines the set of nodes that will keep the old ID and the set of nodes that will get a new ID.

The split interaction between coordinator and participants is showed in Figure8. It is initiated by the coordinator who senses that the number of members is higher than some threshold. It then broadcasts predecessor distance measure

message to all members to find the closest peer who will take charge of forming cliques. The closest peer sends necessary information to other members to partition them into two sets of equal size. After forming two cliques, the coordinator informs its original predecessor and successor to update their links. Step 5 and step 6 shows this process in the case of the predecessor and the successor of the splitting clique is the same.

Also, before performing operations in clique, all members must have a consistent view. Meanwhile, one clique can not join merge and split operation simultaneously. It should process them one by one. Please note that it is unnecessary to update predecessor/successor links when performing merge or split operation since their consistency is kept during the updating of routing table. When some peer receives a link update request message targeting to itself but with a wrong ID, it would reply a new link message to specify the correct one according to its routing table. However, it is better to notify predecessor/successor when merging or splitting cliques to speed up the process of reestablishing consistency.

5.9 DHT

A basic function provided by eQuus network is to construct and maintain a distributed hashtable service for all participated peers. Individual peers are able to use it like normal hashtable. Three kinds of operations (*put*, *get* and *remove*) are supported to achieve the service. When a peer wants to publish some data, it first associates the data with an ID which is usually hashed from a consistent hash function such as SHA-1 and transformed into eQuus ID format. Then, the peer routes a *put* message to the destination clique where the data is stored finally. The processes of *get* and *remove* operations are similar to that of *put* operation.

6 QP Encapsulation over TCP/IP

This clause defines the encapsulation techniques that shall be used to carry QP packets between peers in eQuus network. The encapsulation is known as *QP over TCP/IP*. At present, QP encapsulation encapsulations are described for TCP and UDP. The QP encapsulation used with TCP can be applied to the communication between peers which lie in the same clique. The QP encapsulation used with UDP can be applied to the communication between peers which lie in different cliques. Although there is minor encapsulation difference for TCP and UDP, communication protocols should distinguish the two usages to ensure robustness.

6.1 Transmission and Representation of Octets

All QP PDUs consist of an integral number of octets, numbered starting from 1 and increasing in the order that they are put into a TCP/UDP packet. The bits in each octet are numbered from 1 to 8, where 1 is the low-order bit.

When consecutive octets are used to represent a binary number, the lower numbered octet contains the more significant bits of the binary number[9].

When the encoding of (an element of) an QP PDU is represented using a diagram in this clause, the following representations are used:

- Octet 1 is shown toward the top of the page, higher numbered octets being toward the bottom.
- Where more than one octet appears on a given line, octets are shown with the lowest numbered octet to the left, higher numbered octets being to the right.
- Within an octet, bits are shown with bit 8 to the left and bit 1 to the right.

6.2 QP Packet Format for TCP

A summary of the form of a QP packet for TCP is shown in Table1, starting with the protocol version. The fields shown in the table are defined in 6.4.

Field	**Octet Number**
Protocol Version (6.4.2)	1
Packet Type (6.4.3)	2
Packet Body Length (6.4.4)	3-4
Packet Body (6.4.5)	5-N

Table 1: QP Packet Format for TCP

6.3 QP Packet Format for UDP

A summary of the form of a QP packet for UDP is shown in Table2, starting with the protocol identifier. The fields shown in the table are defined in 6.4.

6.4 QP PDU Field and Parameter Definitions

6.4.1 Protocol Identifier

This field is two octets in length, taken to represent an unsigned binary number. Its value determines the uniqueness of a packet transmitted on UDP. The wrap

[9] It is also known as big-endian order used in network communication.

Field	Octet Number
Protocol Identifier (6.4.1)	1-2
Protocol Version (6.4.2)	3
Packet Type (6.4.3)	4
Packet Body Length (6.4.4)	5-6
Packet Body (6.4.5)	7-N

Table 2: QP Packet Format for UDP

around and reliable problems can be avoided due to the fact that during inter clique communication QP is only used to transfer control messages which belong to typical request/response model.

6.4.2 Protocol Version

This field is one octet in length, taken to represent a dotted version number consisting of two 4-bit unsigned binary numbers. Its value identifies the version of the protocol supported by the sender of the QP packet. An implementation conforming to this specification shall use the value 0001 0000 in this field which means version 1.0.

6.4.3 Packet Type

This field is one octet in length, taken to represent an unsigned binary number. Its value determines the type of packet being transmitted. The following types are defined:

- Route-Req A value of 0000 0000 indicates that the packet carries a request for next hop route.
- Route-Resp A value of 0000 0001 indicates that the packet carries a response for next hop route request.
- Route-Update-Req A value of 0000 0010 indicates that the packet carries a request for updating routing table.
- Route-Update-Resp A value of 0000 0011 indicates that the packet carries a response for updating routing table with some fresh peers.
- Measure-Dist-Req A value of 0000 0100 indicates that the packet carries a request for measuring distance. It can be used to detect liveness. (unicast or groupcast packet)
- Measure-Dist-Resp A value of 0000 0101 indicates that the packet carries a response for measuring distance.

- Join-Query A value of 0000 0110 indicates that the packet carries a request for sampling peers in each routing table.

- Join-Reply A value of 0000 0111 indicates that the packet carries a response for sampling peers.

- Join-Req A value of 0000 1000 indicates that the packet carries a request for joining a clique.

- Join-Resp A value of 0000 1001 indicates that the packet carries a response for join request.

- Member-View-Flush A value of 0000 1010 indicates that the packet carries a changing member view command. (groupcast packet)

- Merge-Req A value of 0000 1011 indicates that the packet carries a request for merging two cliques. A peer who receives such message should forward to other members in the same clique.

- Merge-Req-Forward A value of 0000 1100 indicates that the packet carries a request for merging two cliques forwarded by some other member. (groupcast packet)

- Merge-Resp A value of 0000 1101 indicates that the packet carries a response for merge request.

- Merge-Complete A value of 0000 1110 indicates that the packet carries a merging command sent by coordinator to finish the merge operation. (groupcast packet)

- Pred-Dist-Req A value of 0000 1111 indicates that the packet carries a request for measurements of the distance to the predecessor clique. (groupcast packet)

- Pred-Dist-Resp A value of 0001 0000 indicates that the packet carries a response for predecessor distance measurements.

- Split-Form-Cliques A value of 0001 0001 indicates that the packet carries a request for splitting the clique. The peer who receives such message is then responsible for the actual operation.

- Split-Partition A value of 0001 0010 indicates that the packet carries a request for partitioning the clique according to the set of members which keep their IDs. (groupcast packet)

- Update-Pred-Req A value of 0001 0011 indicates that the packet carries a request for updating the predecessor link of the clique. A peer who receives such message should forward to other members in the same clique.
- Update-Pred-Forward A value of 0001 0100 indicates that the packet carries a request for updating the predecessor link forwarded by some other member. (groupcast packet)
- Update-Succ-Req A value of 0001 0101 indicates that the packet carries a request for updating the successor link of the clique. A peer who receives such message should forward to other members in the same clique.
- Update-Succ-Forward A value of 0001 0110 indicates that the packet carries a request for updating the successor link forwarded by some other member. (groupcast packet)
- Operate-DHT-Req A value of 0001 0111 indicates that the packet carries a request for operating data in the DHT. Depending on the type of the operation in the packet, a peer who receives such message may forward to other members in the same clique.
- Operate-DHT-Forward A value of 0001 1000 indicates that the packet carries a request for operating data in the DHT forwarded by some other member.
- Operate-DHT-Resp A value of 0001 1001 indicates that the packet carries a response for DHT data operation.

6.4.4 Packet Body Length

This field is two octets in length, taken to represent an unsigned binary number. The value of this field defines the length in octets of the Packet Body field (6.4.5); a value of 0 indicates that there is no Packet Body field present.

6.4.5 Packet Body

The Packet Body field is present if the Packet Body Length is greater than 0. The grammar to describe packet body is augmented BNF which is referred in Appendix B. The following rules are used in packet body to describe basic parsing constructs. The US-ASCII coded character set is defined by ANSI X3.4-1986 [1].

```
OCTET          = <any 8-bit sequence of data>
CHAR           = <any US-ASCII character (octets 0 - 127)>
```

```
UPALPHA         = <any US-ASCII uppercase letter "A".."Z">
LOALPHA         = <any US-ASCII lowercase letter "a".."z">
ALPHA           = UPALPHA | LOALPHA
DIGIT           = <any US-ASCII digit "0".."9">
CTL             = <any US-ASCII control character
                (octets 0 - 31) and DEL (127)>
CR              = <US-ASCII CR, carriage return (13)>
LF              = <US-ASCII LF, linefeed (10)>
SP              = <US-ASCII SP, space (32)>
HT              = <US-ASCII HT, horizontal-tab (9)>
<">             = <US-ASCII double-quote mark (34)>
```

eQuus specification defines the sequence `CR LF` as the end-of-line marker for all packet body elements. Packet body is organized by field lines. Each line is separated by an end-of-line marker. The field content is a group of key-value pairs in which the format of value is defined according to Base64 [4]. The actual encoding meaning of a Base64 representation is specified in the field header. If not, a default encoding type `UTF8` [6] SHOULD be used to interpret as a string. Another typical encoding type is `QID` which means an eQuus ID. A field line can only have exact one encoding type. To encapsulate different encoding contents of the same field, the field can be partitioned into two fields with the same name. This mechanism guarantees that the whole packet body is encoded in readable ASCII format.

```
CRLF            = CR LF
SEPARATORS      = "(" | ")" | "<" | ">" | "@"
                | "," | ";" | ":" | "\" | <">
                | "/" | "[" | "]" | "?" | "="
                | "{" | "}" | SP | HT
TOKEN           = *<any CHAR except CTLs or SEPARATORS>
PACKET-BODY     = *(FIELD CRLF)
FIELD           = FIELD-NAME [ "[" ENCODING "]" ] ":" [ FIELD-CONTENT ]
FIELD-NAME      = TOKEN
ENCODING        = "QID"      ; the eQuus ID type
                | "UTF8"     ; UTF8 String representation
                | "ADDR"     ; physical address type
                | "RTAB"     ; routing table
                | <any application data encoding type>
FIELD-CONTENT   = KEY-VALUE-PAIR *( ";" KEY-VALUE-PAIR )
KEY-VALUE-PAIR  = KEY "=" VALUE
KEY             = TOKEN
```

```
VALUE           = Base64 encoding
```

The following paragraphs demonstrate the specific contents of all types of packet body. For simplicity and flexibility, the Base64 encoding of `UTF8` String is described in its normal format and the structures of `QID`, `ADDR` and `RTAB` are up to implementation. Application data encoding types are specified by application protocols built on eQuus protocol. eQuus implementation MAY ignore them when applications running on two peers are not compatible.

Route-Req In a packet carrying a type of Route-Req, its body contains a Request-ID field which is used for resolving its responsible clique. The format of a packet body is:

```
Request-ID[QID] : id=QID Base64 encoding CRLF
```

Route-Resp In a packet carrying a type of Route-Resp, its body can contain a Status field to indicate the type of the response, a Response-ID to indicate the clique to contact or responsible and a set of peer addresses associated with the clique. The format of a success packet body is:

```
Status[UTF8] : type=done CRLF
Response-ID[QID] : id=QID Base64 encoding CRLF
Peer-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
```

The format of a failure packet body is:

```
Status[UTF8] : type=error;desc=wrong format CRLF
```

The format of a redirect packet body is:

```
Status[UTF8] : type=redirect CRLF
Response-ID[QID] : id=QID Base64 encoding CRLF
Peer-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
```

Route-Update-Req In a packet carrying a type of Route-Update-Req, its body contains a Host-ID to indicate the source information, a Prefix to indicate what route entry is currently updated and a Link-ID to indicate the link of the entry. The format of a packet body is:

```
Host-ID[QID] : id=QID Base64 encoding CRLF
Prefix[QID] : prefix=QID Base64 encoding CRLF
Link-ID[QID] : id=QID Base64 encoding CRLF
```

Route-Update-Resp In a packet carrying a type of Route-Update-Resp, its body can contain a Status field to indicate the type of the response, a Prefix to indicate what route entry is currently updated, a new Link-ID if the old link is out of date and a set of peer addresses associated with the clique. The format of a link ok packet body is:

```
Status[UTF8] : type=ok CRLF
Prefix[QID] : prefix=QID Base64 encoding CRLF
Peer-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
```

The format of a link new packet body is:

```
Status[UTF8] : type=new CRLF
Prefix[QID] : prefix=QID Base64 encoding CRLF
Link-ID[QID] : id=QID Base64 encoding CRLF
Peer-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
```

The format of a link lost packet body is:

```
Status[UTF8] : type=lost CRLF
Prefix[QID] : prefix=QID Base64 encoding CRLF
```

Measure-Dist-Req In a packet carrying a type of Measure-Dist-Req, its body contains no content.

Measure-Dist-Resp In a packet carrying a type of Measure-Dist-Resp, its body contains no content.

Join-Query In a packet carrying a type of Join-Query, its body contains no content.

Join-Reply In a packet carrying a type of Join-Reply, its body contains a set of sampling addresses in the request peer's routing table. The format of a packet body is:

```
Peer-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
```

Join-Req In a packet carrying a type of Join-Req, its body contains no content.

Join-Resp In a packet carrying a type of Join-Resp, its body contains the ID of the clique, a set of addresses of all other clique members, the routing table and the application specific data items. The format of a packet body is:

```
Clique-ID[QID] : id=QID Base64 encoding CRLF
Member-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                        addr=ADDR Base64 encoding CRLF
Routing-Table[RTAB] : rtable= RTAB Base64 encoding CRLF
App-Data[<data encoding>] : app data representation CRLF
```

Member-View-Flush In a packet carrying a type of Member-View-Flush, its body contains the host address information, the old view identifier, a set of old member addresses, a set of message sequence numbers recorded by the host, the new view identifier and a set of new member addresses. The key part of the sequence number set is exactly the value part of the old view address set. The format of a packet body is:

```
Host-Addr[ADDR] : addr=ADDR Base64 encoding CRLF
Old-View-ID[UTF8] : id=UTF8 Base64 encoding CRLF
Old-View-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                          addr=ADDR Base64 encoding CRLF
Old-View-Seq-Set[UTF8] : ADDR Base64 encoding=Seq;
                         ADDR Base64 encoding=Seq; CRLF
New-View-ID[UTF8] : id=view ID Base64 encoding CRLF
New-View-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                          addr=ADDR Base64 encoding CRLF
```

Merge-Req In a packet carrying a type of Merge-Req, its body contains the host clique ID, a set of member addresses, the successor clique ID and its associated peer addresses, and the application specific data items. The format of a packet body is:

```
Host-ID[QID] : id=QID Base64 encoding CRLF
Member-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                        addr=ADDR Base64 encoding CRLF
Succ-ID[QID] : id=QID Base64 encoding CRLF
Succ-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
App-Data[<data encoding>] : app data representation CRLF
```

If the destination peer is not the coordinator in its clique, the peer should resend this packet to the coordinator with appending source peer address in the packet body. The format of a resent packet body is:

```
Src-Addr[ADDR] : addr=ADDR Base64 encoding CRLF
Host-ID[QID] : id=QID Base64 encoding CRLF
Member-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                        addr=ADDR Base64 encoding CRLF
Succ-ID[QID] : id=QID Base64 encoding CRLF
Succ-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
App-Data[<data encoding>] : app data representation CRLF
```

Merge-Req-Forward In a packet carrying a type of Merge-Req-Forward, its body contains the same as that of normal Merge-Req.

Merge-Resp In a packet carrying a type of Merge-Resp, its body can contain a Status field to indicate the type of the response, a Host-ID to identify the sender clique and its members, the predecessor clique ID and its associated peer addresses, the routing table and the application specific data items. The format of a successful packet body is:

```
Status[UTF8] : type=ok CRLF
Host-ID[QID] : id=QID Base64 encoding CRLF
Member-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                        addr=ADDR Base64 encoding CRLF
Pred-ID[QID] : id=QID Base64 encoding CRLF
Pred-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
Routing-Table[RTAB] : rtable= RTAB Base64 encoding CRLF
App-Data[<data encoding>] : app data representation CRLF
```

The format of a failure packet body is:

```
Status[UTF8] : type=error;desc=sync CRLF
Host-ID[QID] : id=QID Base64 encoding CRLF
```

Merge-Complete In a packet carrying a type of Merge-Complete, its body contains the same as that of successful Merge-Resp.

Pred-Dist-Req In a packet carrying a type of Pred-Dist-Req, its body contains no content.

Pred-Dist-Resp In a packet carrying a type of Pred-Dist-Resp, its body contains the sender's address and its distance measurement in millisecond unit. The format of a packet body is:

```
Host-Addr[ADDR] : addr=ADDR Base64 encoding CRLF
Dist[UTF8] : dist=100 CRLF
```

Split-Form-Cliques In a packet carrying a type of Split-Form-Cliques, its body contains no content.

Split-Partition In a packet carrying a type of Split-Partition, its body contains a set of addresses to indicate which member is keeping its clique ID. The format of a packet body is:

```
Keep-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
```

Update-Pred-Req In a packet carrying a type of Update-Pred-Req, its body contains a Host-ID to identify the sender clique, the new predecessor clique ID and its associated peer addresses. The format of a packet body is:

```
Host-ID[QID] : id=QID Base64 encoding CRLF
Pred-ID[QID] : id=QID Base64 encoding CRLF
Pred-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
```

Update-Pred-Forward In a packet carrying a type of Update-Pred-Forward, its body contains the same as that of Update-Pred-Req.

Update-Succ-Req In a packet carrying a type of Update-Succ-Req, its body contains a Host-ID to identify the sender clique, the new successor clique ID and its associated peer addresses. The format of a packet body is:

```
Host-ID[QID] : id=QID Base64 encoding CRLF
Succ-ID[QID] : id=QID Base64 encoding CRLF
Succ-Addr-Set[ADDR] : addr=ADDR Base64 encoding;
                      addr=ADDR Base64 encoding CRLF
```

Update-Succ-Forward In a packet carrying a type of Update-Succ-Forward, its body contains the same as that of Update-Succ-Req.

Operate-DHT-Req In a packet carrying a type of Operate-DHT-Req, its body contains a type field to indicate the operation, a data ID and possibly associated value. The format of a *get* packet body is:

```
Operation[UTF8] : type=get CRLF
Data-ID[QID] : id=QID Base64 encoding CRLF
```

The format of a *put* packet body is:

```
Operation[UTF8] : type=put CRLF
Data-ID[QID] : id=QID Base64 encoding CRLF
App-Data[<data encoding>] : app data representation CRLF
Data-ID[QID] : id=QID Base64 encoding CRLF
App-Data[<data encoding>] : app data representation CRLF
```

The format of a *remove* packet body is:

```
Operation[UTF8] : type=remove CRLF
Data-ID[QID] : id=QID Base64 encoding CRLF
App-Data[<data encoding>] : app data representation CRLF
Data-ID[QID] : id=QID Base64 encoding CRLF
App-Data[<data encoding>] : app data representation CRLF
```

Operate-DHT-Forward In a packet carrying a type of Operate-DHT-Forward, its body contains the same as that of Operate-DHT-Req.

Operate-DHT-Resp In a packet carrying a type of Operate-DHT-Resp, its body contains a type of operation, a status field and the result. The format of a *get* packet body is:

```
Operation[UTF8] : type=get CRLF
Status[UTF8] : type=ok CRLF
Data-ID[QID] : id=QID Base64 encoding CRLF
App-Data[<data encoding>] : data=app data representation;
                            data=app data representation CRLF
```

The format of a *put* or a *remove* packet body is:

```
Operation[UTF8] : type=get CRLF
Status[UTF8] : type=ok CRLF
```

A View-Synchronous Reliable Broadcast

A view-synchronous reliable broadcast protocol also delivers the views to the application. This implementation (like many practical ones) must be able to *block* the application during view changes so that it does not $v - send$ any messages for some time. Messages among all pairs of servers are sent over reliable point-to-point links with FIFO delivery. Here is the algorithm for P_i:

initialization:

$s \leftarrow 0$ //P_i's sequence number

// sequence number of last *v-delivered* message from P_j

$s_j \leftarrow 0 \ \forall j \in [1, n]$

$vid \leftarrow 0; view \leftarrow \{P_i\}$ //current view

$new_vid \leftarrow 0; new_view \leftarrow \emptyset$ //next view while it is being installed

upon *v-send(m)*:

send message $(\texttt{send}, vid, s, m)$ to all servers

$s \leftarrow s + 1$

upon *receiving a message* $(\texttt{send}, v, s', m)$ from P_j with $v = vid$:

remember (j, s', m) as a message delivered in view vid

if $(new_vid = 0)$ **or** $(new_vid \neq 0$ **and** $P_j \in view \cap new_view)$ **then**

v-deliver(m)

$s_j \leftarrow s'$

upon *v-change(v, V)*:

$new_vid \leftarrow v; new_view \leftarrow V$

send message $(\texttt{flush}, new_vid, i, view, (s_1, \ldots, s_n))$ to all servers

block the application

upon *receiving* msgs. $(\texttt{flush}, v, j, view', (s'_1, \ldots, s'_n))$ with $v = new_view$ and $view' = view$ from all $P_j \in view \cap new_view$:

for each $P_l \in view$ **do**

$t_l \leftarrow$ maximum of the received s'_l values

if $s_l < t_l$

v-deliver all messages from P_l up to sequence number t_l;

recover the missing messages from other members of new_view who have remembered them

output *v-change(new_vid, new_view)*

$vid \leftarrow new_vid; view \leftarrow new_view$

$new_vid \leftarrow 0; new_view \leftarrow \perp$

unblock the application

If the group is stable, then the membership service will install the same view at all group members. Hence, all members who transition together from the same view to the new view compute the same *cut*, i.e., the set of maximal sequence numbers t_l for $P_l \in view$. Therefore, they *v-deliver* the same set of messages in *view* before installing *new_view*. Because all $P_j \in new_view$ receive the same *v-change* event, they all send a flush message for *new_view*. Hence, P_i eventually receives such messages from all members of the new view and installs the new view.

The set of remembered messages has to be garbage-collected eventually. For knowing when it is safe to forget the delivered messages, a separate "stability mechanism"is usually employed. This is a protocol that periodically informs all servers about the messages delivered by the others.

B Augmented BNF

All of the mechanisms specified in packet body section are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [3]. Implementors will need to be familiar with the notation in order to understand this specification. The augmented BNF includes the following constructs:

name = definition
The name of a rule is simply the name itself (without any enclosing "<" and ">") and is separated from its definition by the equal "="character. White space is only significant in that indentation of continuation lines is used to indicate a rule definition that spans more than one line. Certain basic rules are in uppercase, such as SP, LWS, HT, CRLF, DIGIT, ALPHA, etc. Angle brackets are used within definitions whenever their presence will facilitate discerning the use of rule names.

"literal"
Quotation marks surround literal text. Unless stated otherwise, the text is case-insensitive.

rule1 | rule2
Elements separated by a bar ("|") are alternatives, e.g., "yes | no"will accept yes or no.

(rule1 rule2)
Elements enclosed in parentheses are treated as a single element. Thus, "(elem (foo | bar) elem)"allows the token sequences "elem foo elem"and "elem bar elem".

*rule
The character "*"preceding an element indicates repetition. The full form is "<n>*<m>element"indicating at least <n> and at most <m> occurrences of element. Default values are 0 and infinity so that "*(element)"allows any number, including zero; "1*element"requires at least one; and "1*2element"allows one or two.

[rule]
Square brackets enclose optional elements; "[foo bar]"is equivalent to "*1(foo bar)".

N rule

Specific repetition: "<n>(element)"is equivalent to "<n>*<n>(element)"; that is, exactly <n> occurrences of (element). Thus 2DIGIT is a 2-digit number, and 3ALPHA is a string of three alphabetic characters.

#rule

A construct "#"is defined, similar to "*", for defining lists of elements. The full form is "<n>#<m>element"indicating at least <n> and at most <m> elements, each separated by one or more commas ("","") and OPTIONAL linear white space (LWS). This makes the usual form of lists very easy; a rule such as

```
( *LWS element *( *LWS "," *LWS element ))
```

can be shown as

```
1#element
```

Wherever this construct is used, null elements are allowed, but do not contribute to the count of elements present. That is, "(element), , (element)"is permitted, but counts as only two elements. Therefore, where at least one element is required, at least one non-null element MUST be present. Default values are 0 and infinity so that "#element"allows any number, including zero; "1#element"requires at least one; and "1#2element"allows one or two.

; comment

A semi-colon, set off some distance to the right of rule text, starts a comment that continues to the end of line. This is a simple way of including useful notes in parallel with the specifications.

implied *LWS

The grammar described by this specification is word-based. Except where noted otherwise, linear white space (LWS) can be included between any two adjacent words (token or quoted-string), and between adjacent words and separators, without changing the interpretation of a field. At least one delimiter (LWS and/or separators) MUST exist between any two tokens (for the definition of "token"below), since they would otherwise be interpreted as a single token.

References

[1] ANSI. Coded Character Set - 7-Bit American Standard Code for Information Interchange, 1986.

[2] S. Bradner. Key words for use in RFCs to Indicate Requirement Levels. RFC 2119 (Best Current Practice), Mar. 1997.

[3] D. Crocker. STANDARD FOR THE FORMAT OF ARPA INTERNET TEXT MESSAGES. RFC 822 (Standard), Aug. 1982. Obsoleted by RFC 2822, updated by RFCs 1123, 2156, 1327, 1138, 1148.

[4] S. Josefsson. The Base16, Base32, and Base64 Data Encodings. RFC 4648 (Proposed Standard), Oct. 2006.

[5] T. Locher. equus: A provably robust and efficient peer-to-peer system. Master's thesis, ETH Zurich, 2005.

[6] F. Yergeau. UTF-8, a transformation format of ISO 10646. RFC 2279 (Draft Standard), Jan. 1998. Obsoleted by RFC 3629.