Load Sharing for Mobile Streaming Services using LIN6

Hui Tiong Khoo
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara, Japan 630-0101
jonath-k@is.aist-nara.ac.jp

Mitsunobu Kunishi
Keio University, Japan
kunishi@tokoro-lab.org

Masahiro Ishiyama
Communication Platform Laboratory, R&D Center, Toshiba Corporation
masahiro@isl.rdc.toshiba.co.jp

Hideki Sunahara
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara, Japan 630-0101
suna@wide.ad.jp

Abstract - We envisage that most of the subway train services or roadways in the near future will provide wireless coverage to commuters. This will drive the need for content providers to offer seamless mobile web services like streaming video etc. for mobile computers and handheld devices. We see a huge potential in providing the next generation services based on IPv6 as the larger pool of network addresses provides for real end-to-end communications. Although this can be achieved using Mobile IPv6 (MIPv6), it has several features which can be improved upon. Location Independent Network Architecture for IPv6 (LIN6) has been proposed as an alternative. The advantages include lesser packet overheads and the ability to house its Mapping Agents across different networks thus avoiding a single point of failure when the home network goes down. In this paper, we propose a load sharing architecture of a server pool providing mobile streaming services using LIN6 addressing. We address the issue of the latency between the mobile nodes and the Mapping Agents. We also show how we can allocate the required bandwidth to the end-users with their subscribed level of service. Finally, we present a comparative measurement study between MIPv6 and LIN6 and showed that LIN6 is indeed better in terms of data transfer performance.

Keywords
Mobile Streaming Services, Load Sharing

I. INTRODUCTION

The proliferation of PDAs, cellular phones and mobile notebooks has propelled the demand for more mobile web services like online news clip streaming from CNN etc. The collapse of the Internet bubble may have slowed the amount of investments pouring into this area of development, but it cannot halt the growth in the demand for such services. On one end, we have the cellular phones and PDA manufacturers churning out faster and better products with wireless and Bluetooth connections. On the other end of the spectrum, we have the rapidly burgeoning cellular phone users growing by the day. It has become a necessity for telecommunication companies to convert their circuit switch network infrastructure to a packet switch network infrastructure for them to save cost (as data traffic outstrips voice traffic) and to leverage on the Internet to provide more value added services to their customers. However, the take up rate of new cellular phones and wireless PDA also depends on the availability of interesting content for the end users. Thus, sooner or later, we will see the growth of companies providing mobile web services.

We are concerned with the continuing availability of such web services even when we are moving across networks. An example is watching a news video clip in the subways via a wireless connection or making a call using VoIP. For streaming applications and services, it is a must that we use a mobile network protocol to allow the streaming to continue as we move across different network segments. Mobile IPv6 (MIPv6) [2] is a mobile protocol designed for macro-mobility in an IPv6 environment. However MIPv6 has additional packet overheads, which could be extremely costly if the average packet size is small.

LIN6 [1] addresses are similar to the normal IPv6 addresses and we have Mapping Agents (MA) which are synonymous to the Home Agents of MIPv6.

In this paper, we outline an architecture based on LIN6 addressing which provides for load sharing amongst distributed servers and bandwidth allocation depending on the subscribed level of service by the end users. The performances of the mobile nodes can also be monitored from a central location which will allow the administrator to detect any network troubles.

The rest of the paper is structured as follows. Section II provides an introduction of LIN6. Section III describes our architecture. Section IV talks about implementation details. Section V presents a comparison of the measurements we took. Section VI contains the conclusions and Section VII describes our future work.

II. LIN6

A. MIPv6 and LIN6

From a protocol design point of view, any mobile protocol will need to do address mapping at some point of time. MIPv6 does this at the receiving node. Thus the sending node has to send the necessary information across the network. This is why MIPv6 needs the Home Address Extension Header (24 bytes) and Type 2 Routing Header (20 bytes). LIN6 does the mapping at both the sending node and receiving node. By knowing beforehand the format of the original address, it saves the need to send the extra information along with the packet. The obvious advantage is a smaller packet size. However, in order for LIN6 kind of mobile protocol to work, we need a way of differentiating the address before we can do the mapping. The protocol designers for LIN6 introduce the LIN6 prefix and LIN6 Node ID to help differentiate between a LIN6 address and a normal IPv6 address.

B. LIN6 Address

A LIN6 address, similar to an IPv6 address, is divided into two portions – an upper 64-bit fixed LIN6 prefix and a lower 64-bit Node ID. The LIN6 prefix is a fixed non-routable value of 3ffe:501:1830:1999. The LIN6 Node ID conforms to the EUI-64 format [11] and the upper 24 bits of the Node ID is represented by a fixed Organizationally Unique Identifier (OUI) assigned by IEEE. The OUI value is 00:01:4a. A LIN6 locator is made up of an upper 64 bits routable network prefix and a lower 64 bits Node ID. Mobility is achieved by mapping the LIN6 address to the LIN6 locator before the packet leaves the node.

We used the global bit in the Node ID to differentiate between a stationary and a mobile node. A stationary node uses its current network prefix as the upper 64 bits and the Node ID has its global bit set to 0. An example of a stationary node LIN6 address is 3ffe:501:808:4001:1:4a00::1. An example of a mobile node LIN6 address is 3ffe:501:1830:1999:201:4a00::2. Note that the upper 24 bits for a stationary Node ID is 00:01:4a and that for mobile Node ID is 02:01:4a.

At the application layer, LIN6 addresses are used for communication between applications (see Fig 1). The LIN6 addresses are mapped onto the LIN6 locators before the packet is sent from the node. On the receiving end, the LIN6 locators are mapped back into a LIN6 address before the packet is received by the application.

As you can see, we need the LIN6 prefix so that we can have a consistent address that applications can use to communicate between one another as they move across different networks. We also need the OUI value to help us differentiate the addresses for the mapping of LIN6 locators to LIN6 addresses.

III. LIN6 Architecture

Any mobile node that wants to use LIN6 needs to be configured with a LIN6 address first. The user will get his unique Node ID from the administrator and he needs to run a resolution daemon (*lin6resolvd*) to register his node with a MA. The mobile node will report its current routable address together with its Node ID to the MA (see Fig 2). Note that in Fig 2, LIN6 Node 2 gets its MA address in the same manner as LIN6 Node 1. The DNS is not shown in the figure.

The LIN6 address is configured in the DNS by the administrator beforehand. The LIN6 address remains consistent even when the nodes move. The LIN6 locator is the one that changes.

The MA is located by first sending a query to the DNS for the AAAA record. DNS will reply with a LIN6 address which is non-routable. The daemon, *lin6resolvd*, will try with a second query to the DNS for the TXT record of that LIN6 address; this is similar to the reverse query. DNS will reply with the MA address in the TXT record.

For MIPv6, there is no need to query twice for the Home Agent (HA) because the HA is a router located in the Home Network (HN). There can be multiple HAs, but all the HAs must be located in the HN. This is not good because if the HN goes down when the Mobile Node (MN) is in a Foreign Network (FN), there is no way to communicate with the MN. To overcome this problem, LIN6 protocol introduces the concept of distributed MAs. To locate the MA in a reliable and efficient manner, LIN6 propose to use DNS reverse query.

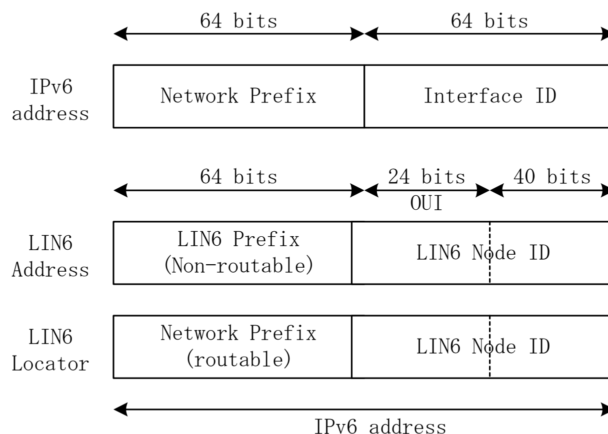

Fig 1. LIN6 Addressing Method

When two LIN6 nodes want to communicate with each other, they will check their own Mapping Table (see Fig 3) (synonymous with the Binding Cache in MIPv6), if the entry cannot be found within the table, the mobile node will then query the MA for the record of the opposing node. Note that Node 1 and 2 can use different MAs.

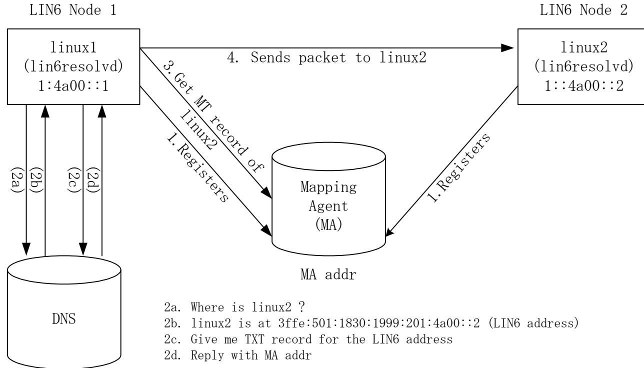

Fig 2. LIN6 Architecture

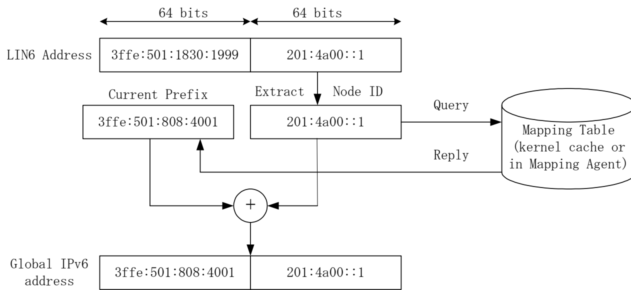

Fig 3. LIN6 Address Mapping

The LIN6 nodes should be administer by a central figure to avoid clashing of Node IDs. This is a major drawback as other entities or organizations have to request their Node ID from a central authority. However, looking from another perspective, there is no need to use the same LIN6 prefix or OUI value at all. Organizations can implement their very own LIN6-like mobile network without any outside interference.

IV. PROPOSED ARCHITECTURE

A. Load Sharing Server

We proposed an architecture consisting of a pool of servers that are distributed across a wide area. The aim is to serve streaming applications. Currently, most content servers redirect users' request to the nearest server with the cached contents. However, the servers will not be able to maintain the connection for streaming if the user's network address changes when he moves into a new area. A likely example is a different network segment for each major train station. Assuming that the user is redirected to a nearby server using MIPv6, he/she still needs to content with the overheads of MIPv6 protocol. Based on the respective subscription rates and schemes, the cost for the packet overheads can be quite hefty.

We consider two different scenarios using LIN6. We can either use multiple LIN6 addresses or just a single address for a pool of servers across a wide area.

For the first scenario, we can have load sharing capabilities by redirecting the user to the nearest server when the user logins through a website. However, we will not be able to address this mobile web service using the same URL for all the servers. The website will be responsible for making an intelligent choice of redirecting the user to the nearest URL. This is the current service model for many websites.

For the second scenario, the choice of selecting the nearest server is made only during address resolution. This makes it possible for the user to address a whole server pool using a single URL. Thus, we favor the second scenario as our solution. Note that this requires modifications on the original LIN6 model. We will cover the details of the various changes in Section IV.

In Section II.B, we mentioned that the mobile node needs to make two queries to find out the location of the MA. We modify this name resolution portion to accommodate our load sharing server into the LIN6 architecture. Instead of returning the MA address in the TXT record, the DNS will now reply with the location of the load sharing server. The DNS can also provide more than one locations for the load sharing server so as to prevent a single point of failure.

We denote each content server as S_i, MA as MA_i and the load sharing server as LS. Each MA_i can be housed in different or the same boxes as S_i. In practice, both the MA and the content server S exist in the same box. Note that the content server S is a LIN6 node. During setup, the daemon for MA (i.e. *lin6agentd*) will have to be run first. Each MA_i will register itself with one of the LS by sending a heartbeat. The MAs can get the address of the LS servers from the DNS using reverse query. After all the MAs have registered themselves, LS will have a list of MAs that are not associated with any content servers yet. Each S_i will then run their name resolution daemon (i.e. *lin6resolvd*), this will query the LS for the most appropriate MA. The S_i will register with the nearest MA_i as we use RTT as the yardstick. Assuming that the content servers are distributed far away from one another, they will register with the MA running in the same box.

We modified the daemon (*lin6resolvd*) such that it will use the same MA_i until the S_i is rebooted. Hence, now we have a tight coupling between the MA_i and S_i. Since all the S_i uses the same LIN6 addresses, we cannot allow any S_i to re-register itself with another MA_i after the initial registration as LS selects MA based on the assumption that MA is matched to a content server S on a one-to-one relationship. After registration, the MA will send periodic heartbeat signal to the LS to indicate that it is still alive. If

the LS did not receive any heartbeat from a particular MA for a certain period, it will mark the MA as inactive.

Assume that we now have a mobile node MN that wishes to use the mobile web service. Before connection, the MN must configure its LIN6 address using its Node ID. This Node ID is assigned by the administrator and it must match the record in the database. Likewise, the MN will query the DNS and get the address of the LS. The LS will reply the MN with the most appropriate MA_i to use. When the MN tries to register with the MA, the MA will check the database for records. No registration is allowed if the Node ID of MN is not in the database (see Fig 4).

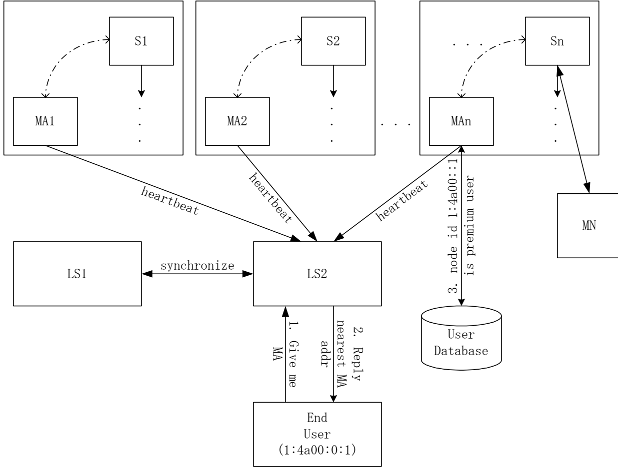

Fig 4. Load Sharing Architecture using LIN6

When MA queries the database for records, it also retrieves the user's subscribed level of service. It then uses the Linux commands *ipchains* [6] to tag IPv6 packets according to the Node ID and *tc* [3] (Linux Traffic Control) to control the bandwidth of the outgoing stream for the packets.

B. Load Sharing Heuristics

The LS carries a load index value for each of the server. This value is calculated whenever a MN connects to the MA. Assume that we have 2 classes of users, normal and premium.

$$load[i] = normal[i] x\ wt_normal + premium[i] x\ wt_premium \quad (1)$$

where normal[i] and premium[i] indicates the number of connections.

The load index value will help the LS to determine whether a server S is capable of satisfying the users subscribed level of service before it assigns the user to S.

The LS will, however, based its decision of choosing MA on the smallest round trip time (RTT) from the MAs to the MN. Since the MAs are tightly coupled to the servers, we can reduce an originally bipartite matching problem to a single dimensional one (i.e. instead of finding the MA with the shortest distance between point A and B, we only need to be concerned with the distance of the MA to point B). The MA that gives the smallest RTT is stored against a value of the network prefix of MN in the LS memory. This will be used to reply for future queries. We associate each entry with a timeout value so that this table can adjust to the dynamic conditions of the network.

Table 1: (MA Table) Time will be reset every time a heartbeat is received. MA is marked as inactive when timer runs out. Entry is not deleted.

MA Address	Load Index	Timer
MA1 addr	100	119
MA2 addr	80	102

Table 2: This is a cache history of the answers based on our heuristics. This helps to cut down the number of queries if they are coming from the same network prefix. Entry is deleted when timer runs out.

Network prefix	MA addr	Timer
3ffe:501:808:4001	3ffe:501:…	120
2001:200:0:3000	3ffe:501:…	101

The algorithm for replying a query with the nearest MA is as follows:

1. When the LS received a query, it first looks at the network prefix of the query. This can be done by using *getpeername*, a function call in c language.
2. It uses the Node ID of the user to query for the user's level of service.
3. LS looks up the network prefix in Table 2 and replies with the MA address if the network prefix is found.
4. LS will increment the Load Index accordingly. It will also notify the MA about the Node ID and the level of service. MA will use this information and ask Server S to tag the packets and set up the necessary bandwidth.
5. If the LS fails to find a cached answer, it will has to go through the heuristic decision procedure.

Heuristic Decision Procedure

1. Get network prefix and Node ID of peer-connection.
2. Get level of service using Node ID.
3. Look up MA Table (see Table 1). Choose those MA who are still alive and Load Index is still sufficient to support the level of service we obtained in 2. We assign a global threshold value (Th_{load}) which we use to compare with the load index so that we can distribute the load equally among the servers.

4. Request MA to do a RTT to our user's address.

5. Reply the user with the MA that has the lowest RTT.

C. Comparison with Linux Virtual Server

Our design resembles the Linux Virtual Server (LVS) [8] to a certain degree, but it has some comparative advantages over LVS.

LVS uses a virtual IP to represent a pool of servers and round robin among them to server incoming requests. The director server, the central figure in LVS, forwards the incoming request to a prospective server within the pool. The director server can be a single point of failure and the suggested remedy is to prepare a backup director server to check for heartbeats from the main director server via a serial cable. The backup director server will attempt to take over the role when it does not hear the heartbeats through gratuitous ARP.

Although LVS can be configured to use a server cluster over the wide area, it does so via IP tunneling which introduces packet overheads (see Fig 5). Furthermore, the director server must be alive throughout the whole lifetime of a single connection. The LVS cannot leverage on the proximity advantage of their servers to the client since all packets have to be redirected from the director. Finally, the director server is a stateful server. All ongoing connection breaks if the director server crashes.

For our design, once the LS server provides the client with a prospective MA server, the client will only need that MA to help maintain its connection. Since the MA is only responsible for mapping addresses, most of the communication happens directly between the client and the server as both keeps a copy of the mapping relationship within their cache (just like Binding Cache). The client will only need to query the MA again if the mapping in the cache runs obsolete.

The only caveat here is that if it happens that the MA is down for some reasons, the client will also loose its connection to the server S when the client's Mapping Table entry is deleted.

The LS is also stateless compare to the director server. In cases when the LS crashed, concurrent connections will not be affected. The disadvantage is that the loss of the connection information will affect the decision making based on the heuristics, but we synchronize the table of a LS server which is rebooted with neighboring LS servers to achieve consistency.

D. Inter-LS servers Communications

It is evident from the above description that the LS server is a single point of failure. To overcome this problem, we introduced a few other LS servers to serve as backups. We accomplish this by adding more entries to the DNS AAAA records. This list of LS servers IPv6 addresses are kept in a configuration file which is read by the LS

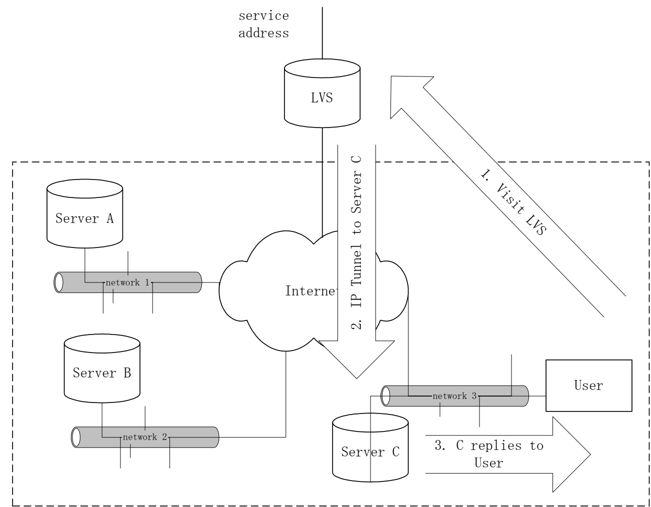

Fig 5. LVS tunneling over wide area. User has tunneling overheads and little proximity advantage.

servers during initialization time. The servers may not all run at the same time, hence a LS server will first keep a list of addresses and attempt to establish connections to the neighboring servers. The server will retry this at a regular interval until it reaches a maximum number of tries.

Each LS server will be connected to (*n-1*) neighboring LS servers. Each MA server can register with any LS server. Each MA will only be associated with one LS server, but the LS server will relay the heartbeat to the neighboring LS servers upon receiving it. The MA must send a regular heartbeat signal to the LS server. If the LS server fails, the MA must search for a new LS server by re-querying the DNS server. On the other hand, if the LS server fails to receive a heartbeat from the MA, it will mark the MA entry in the table as inactive. This will prevent the LS server from giving the inactive MA as a reply to a query from the end users.

When a LS server receives a request for MA from the end users, it will first check its own history table for a reply. We match a reply according to the network prefix of the user. The reply will remain valid for a certain lifetime before we delete it away from the cache. We introduce this feature so that we would not waste bandwidth on extra RTT messaging. If a reply exists, we will use this cached reply. If we cannot find an answer based on the user's network prefix, we will have to use our heuristics to get the best answer.

First, we will look into the MA table (see Table 1) in the LS server. The MA table contains both active and inactive MA entries. We will send a RTT request message to all the entries that are still active and have a load index below threshold. The MA will then calculate the average of four RTT timings to the end user's node and return the result to the LS server. The LS server will then record the reply with the lowest RTT timings in the history table. This answer will be propagated to the rest of the LS servers.

We define 8 types of messages which we are using for LS servers. Messages 1 to 5 are self explanatory. We use message 6 and 7 to relay updates to neighboring LS servers. As for message 8, we use it to synchronize LS server. We mentioned earlier that the LS server will attempt to connect to its neighboring servers during startup. The first message it sends will be LSS_SYNC_REQUEST. This will help to ensure synchronization of tables for all the LS servers. In practice, we would expect to have no more than three LS servers. Each table is also limited to a size of 64 (implying that LS can hold up to 64 MA records).

Table 3: Messaging for LS servers

Server	Message Type	Value
MA to LS	LSS_HEARTBEAT	0x01
	LSS_RTT_REQUEST	0x02
	LSS_RTT_REPLY	0x03
User to LS	LSS_MA_REQUEST	0x04
	LSS_MA_REPLY	0x05
Between LS	LSS_MATBL_UPDATE	0x06
	LSS_ANSTBL_UPDATE	0x07
	LSS_SYNC_REQUEST	0x08

V. IMPLEMENTATIONS

A. Implementations of LIN6 protocol

The LIN6 protocol has been ported from NetBSD 1.5 to Linux kernel 2.4.20. LIN6 is modeled after the Netlink protocol and the LIN6 socket is a message passing system between the kernel and userspace. We used the IPv6 kernel stack from the USAGI project [9] and keep the Mapping Table for LIN6 in the kernel. We created a pseudo device (*linsix0*) to hook the LIN6 address and be responsible for sending Router Solicitation (RS) packets when the MN reattaches itself to a network. Under normal circumstances, the MN will detect the change of network when it receives a new Router Advertisement (RA) packet from the router. We also define a signal to be sent from the kernel to all the listening LIN6 sockets if a new network prefix is advertised. *lin6resolvd* also keeps tab on the NETLINK messages from the kernel (see Fig 6).

We used NETFILTER hooks for IPv6 to modify the incoming and outgoing packets, otherwise known as *sk_buff* data structures in Linux. When the node has switched network, the new network prefix will be registered in the Mapping Table by sending a message from the userspace to the kernel. If the node tries to communicate with another LIN6 node, the kernel will send a message to the userspace daemon, the daemon will then send a message to the registered MA requesting for the Mapping Entry of the other LIN6 node. Once it gets the reply, it will copy it to the kernel layer and thus the outgoing packets can map their LIN6 addresses to real network addresses. All the actual address mappings are done in the NETFILTER hooks (see Fig 7). We use NETFILTER hooks so that we can organize our codes better and also this will allow us to create our own access list in our extension work.

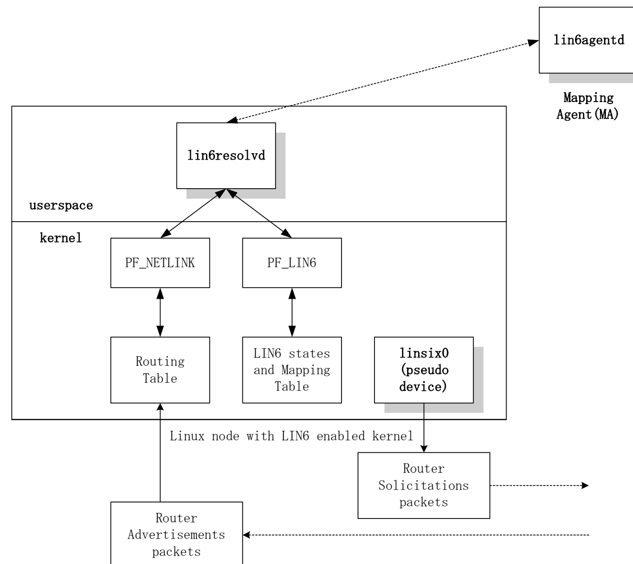

Fig 6. LIN6 implementation on Linux

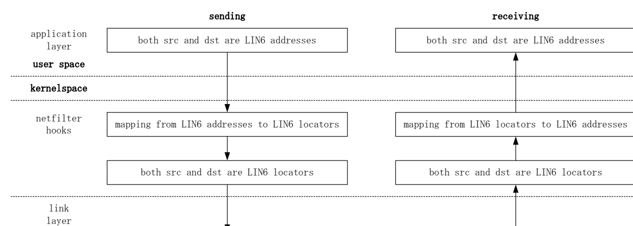

Fig 7. Address Mapping in NETFILTER hooks

B. LIN6 userspace daemons

We need 2 daemons to run the LIN6 protocol. The daemon *lin6resolvd* is required on the mobile nodes and *lin6agentd* is for Mapping Agent. Both are ported over to Linux, most of the codes remain similar except for the route handling portion.

We added route handling calls which listens for RT_NETLINK messages from the kernels and *ioctl* calls to the kernel to modify routes.

C. LS server

LS server is multi-threaded and manages two tables (see Table 1 and Table 2) and three sets of connections. It keeps a list of its neighboring LS servers and connects to them to help synchronize the two tables. It also listens on a port for heartbeats and on another for queries from users asking for MA addresses.

VI. MEASUREMENTS

A. Experiment network

We prepared a small network configuration so that we can get the best performance timing for handover (see Fig 8).

The packet processing timings are recorded from the Athlon 1GHz machine.

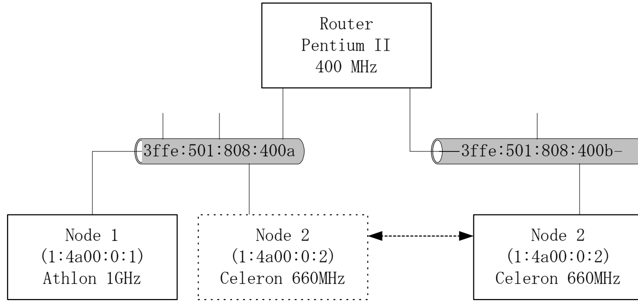

Fig 8. Network configuration

B. Performance comparison between LIN6 and MIPv6

We first compare the packet processing overheads between LIN6 and MIPv6 (MIPL Implementation [10]). We defined processing overheads as the extra processing that is required for the IPv6 packets if MIPv6 or LIN6 is used. In the Linux kernel, we measured the CPU clock cycles needed to process each code segment between the *#ifdef* and *#endif* statements for the 2 separate protocols in the IP layer. We measured for both ICMP (*ping*) and TCP (*ssh*) packets. We noticed that the processing overheads for LIN6 packets are higher than MIPv6 (see Fig 9). This is due to the several *memcpy* statements called in the process of address mapping.

Next we measured the File Transfer Timings for both MIPv6-MIPv6 and LIN6-LIN6 communications for a filesize of 53,850,039 bytes. We found that the performance for LIN6-LIN6 is almost comparable to that for the normal IPv6-IPv6 communications (see Fig 10). The data transfer took longer for MIPv6-MIPv6. This shows that the data transfer timings of packets are more affected by the data size of the packets then the processing time. This also proved our earlier claim that it is advantageous in using LIN6 over MIPv6.

C. Handover Timings Comparisons

Using the same experimental setup, I measured the handover timings for both LIN6-LIN6 and MIPv6-MIPv6 mobility. This is unrealistic as our proposal talks about distributed servers in a Wide-Area Network, however these measurements should give an indications of the performance comparisons between MIPv6 and LIN6.

The switching between the networks is done by disconnecting the notebook PC (Celeron 660 Mhz) and reconnecting it to another network. I measured the time between the reconnection and the time it takes for the next retrieval from the Mapping Table (LIN6) or Binding Cache (MIPv6). For an average of 10 tries, I obtained an average of 199ms for LIN6 and 534 ms for MIPv6 respectively.

Table 4: Processing Overheads (CPU Clock Cycles)

Protocols	ICMP (ping)	TCP (ssh)
LIN6-LIN6	23163	18774
MIPv6-MIPv6	20155	5271

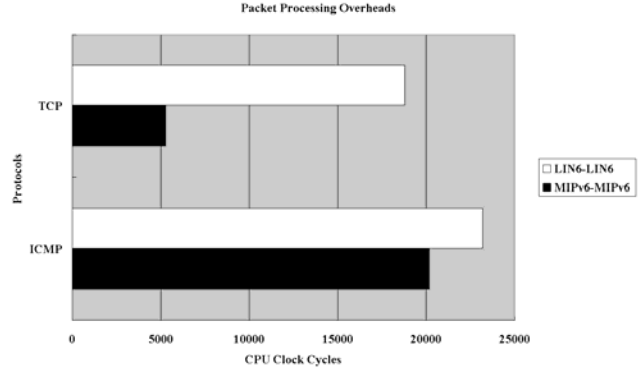

Fig 9. Packet Processing Overheads Comparison

Table 5: File Transfer Timings (Seconds) for 53,850,039 bytes

Protocols	Overheads	Time (Seconds)
IPv6-IPv6	0	70.7
LIN6-LIN6	0	70.8
MIPv6-MIPv6	44	74.9

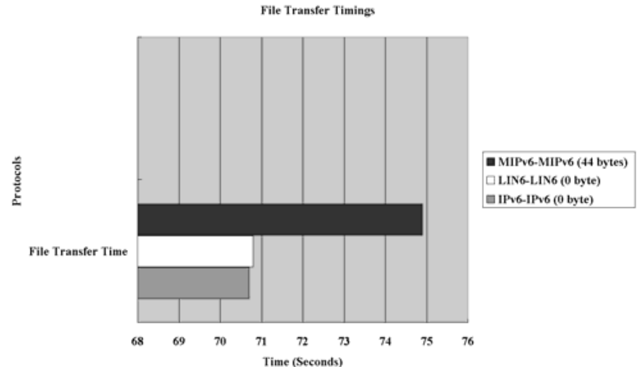

Fig 10. File Transfer Timings Comparison

Although the average timings are different, I notice that there are fluctuations in the handover timings. Sometimes MIPv6 has a better timing than LIN6 and vice versa. One possible cause for these irregular timings is the Duplicate Address Detection mechanism.

VII. CONCLUSIONS

In this paper, we describe a wide-area load sharing model for mobile web services using mobile protocols based on IPv6. An example is video streaming to handheld devices.

We prefer LIN6 over MIPv6 as LIN6 has lesser packet overheads compare to MIPv6 and it also has the ability to place its Mapping Agents across different networks. For MIPv6, the Home Agent of the mobile nodes could be far away from the nearest service server and this increases the hand over timings. Even though the hand over timings can be improved by using Hierarchical MIPv6 [4], we are still left with the packet overhead issue which we have described earlier. In Section III, we show how our model allows LIN6 nodes to use the nearest Mapping Agent and service server. We also do a comparison between the wide-area load sharing of Linux Virtual Server with our model and described how our architecture can fare better than LVS under some circumstances.

We compare the packet processing overheads and the data transfer performance for both MIPv6 packets and LIN6 packets. We also compare the handover timings for both the protocols. The experiments concluded that LIN6 has a better data transfer performance than MIPv6.

VII. FUTURE WORK

Our model can connect the user to the nearest server, but upon connection, he is stuck with this server until he reconnects. We realize that the performance can be further improved if we have the ability to migrate ongoing connections to the service server nearest to the user as he moves along. Connection migration is not possible with LIN6 or even MIPv6. There are, however other projects that explores mobility through connection migration [5] or session mobility [7].

The LIN6 protocol is only compatible with other LIN6 nodes and does not work with other normal IPv6 nodes without a LIN6 kernel patch. We would like to implement full mobile compatibility with IPv6 as part of our future work.

The experimental result only indicates the performance of LIN6 on a micro-scale. We have to use simulation to extract better results for mobility over a wide area. We also need to work on the micro-mobility performance of LIN6 of the Linux implementation. We also plan to add access list support to LIN6 on Linux for security enhancement.

ACKNOWLEDGEMENTS

Our thanks to the USAGI team for their work on IPv6 for Linux. We would also like to show our appreciation to Dr. Kazutoshi Fujikawa who took time to review this paper.

REFERENCES

[1] M. Ishiyama, M. Kunishi, K. Uehara, H. Esaki, F. Teraoka, LINA: A New Approach to Mobility Support in Wide Area Networks, *IEICE Transactions on Communications*, Vol. E84-B, No.8, Aug 2001

[2] D. Johnson, C. Perkins, "Mobility Support in IPv6", http://www.ietf.org/internet-drafts/draft-ietf-mobileip-ipv6-19.txt

[3] A. Kuznetsov, Linux Traffic Control resources, http://www.sparre.dk/pub/linux/tc/

[4] H. Soliman, C. Castelluccia, K. El-Malki, L.Bellier, "Hierarchical Mobile IPv6 mobility management (HMIPv6)", http://www.ietf.org/internet-drafts/draft-ietf-mobileip-hmipv6-07.txt

[5] A. Snoeren, H. Balakrishnan and M.F.Kaashoek "The Migrate Approach to Internet Mobility", Proc. Of the Oxygen Student Workshop, July 2001, http://nms.lcs.mit.edu/~snoeren/papers/migrate-sow.html

[6] R. Russell, Linux IP Firewalling Chains, http://www.netfilter.org/ipchains/

[7] J. Salz, A. Snoeren, "TESLA, The Transparent Extensible Session-Layer Architecture for End-to-End Network Services", http://nms.lcs.mit.edu/tesla/

[8] Linux Virtual Server Project, http://www.linuxvirtualserver.org/

[9] The USAGI Project, http://www.linux-ipv6.org

[10] MIPL Mobile IPv6 for Linux http://www.mipl.mediapoli.com/

[11] Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority,http://standards.ieee.org/regauth/oui/tutorials / EUI64.htm