Design and Implementation of TCPHA

(Draft Release)

Li Wang

August 2005

Table of Contents

Table of Contents 2
Table of Pictures 3
Abstract 4
1 System Function and Characteristics 5
2 System Architecture 6
3 Working Principle 7
4 Key Techniques 9
4.1 Handoff Protocol 9
4.2 TCP Handoff implementation 10
4.2.1 Connection Reconstructing 11
4.2.2 HTTP Request Zero Copy Relay 12
4.2.3 Connection Relay 12
4.3 Symmetric Multiple-Thread Transaction-Driven Architecture 13
4.4 ARP problem and the solution: ARP Filtering 14
4.5 Regular Expression Rule Matching 15
4.6 Dynamic IP Tunneling 15
4.7 Local Node Feature 16
4.8 P-HTTP Support (Under Discussion) 16
4.8.1 Single Handoff Course 16
4.8.2 Multi-Handoff Course 17
4.8.3 BE Scheduling Technique 18
4.9 High Availability 20
4.10 Dynamic Scalability 20
4.11 Journaling 20

Table of Pictures

Figure 1. 1 TCPHA's Goal 5
Figure 2. 1 TCPHA architecture 6
Figure 3. 1 TCPHA packets flow 8
Figure 4. 1 Handoff request 9
Figure 4. 2 Handoff ACK 9
Figure 4. 3 TCP handoff implementation 11
Figure 4. 4 Linux network data structure 12
Figure 4. 5 TCPHA process flow 13

Abstract

In popular cluster-based Web server scheduling techniques, content-aware scheduling is a popular scheduling technique, it has many advantages. But the existing systems have low scalability, scheduler performance bottleneck problem, which lead to the advantages can not be fully exerted. TCP Handoff is a novel core technique to support content-aware scheduling, but for the implementation is difficult, at present the technique is still under discussion, no open source implementation on Linux and no TCP Handoff based practical cluster scheduling systems appear. We propose a TCP Handoff implementation, compare with the popular implementation techniques, it has higher performance. It adopts some novel techniques, such as connection reconstruction, connection relay and HTTP zero copy relay. ARP filtering is an optimal solution to ARP problem. Based these techniques, and dynamic IP tunnel, multi-handoff, we propose a novel content-aware scheduling system TCPHA. It runs inside the OS kernel, avoids the overhead of context switching and memory copying between user-space and kernel-space, has high performance. It is implemented as a loadable kernel device driver module, no need modifying network stack. No need modifying user space server application codes and browser codes, everything is transparent from user space perspective and from client perspective. The system installation and configuration is very simple. System also supports regular expression, the administrator can set very complicated schedule rules. TCPHA can be used to build a high performance and high availability server based on a cluster of Linux servers. Such as cluster of big website, especially the software download website and media service website. Furthermore, with some tiny modifications, the core techniques can be used in distributed computing, fault tolerance, fault recovery, backup fields etc. TCPHA has been published in internet (http://dragon.linux-vs.org/~dragonfly/) and is attracting more attentions. It has been accepted by well-known LVS project as a subproject of it.

1 System Function and Characteristics

TCPHA can be used to build a high-performance and high available server based on a cluster of Linux servers. TCPHA implements kernel scalable content-aware request distribution based on TCP Handoff for the Linux operating system. The function of TCPHA is illustrated as follows:

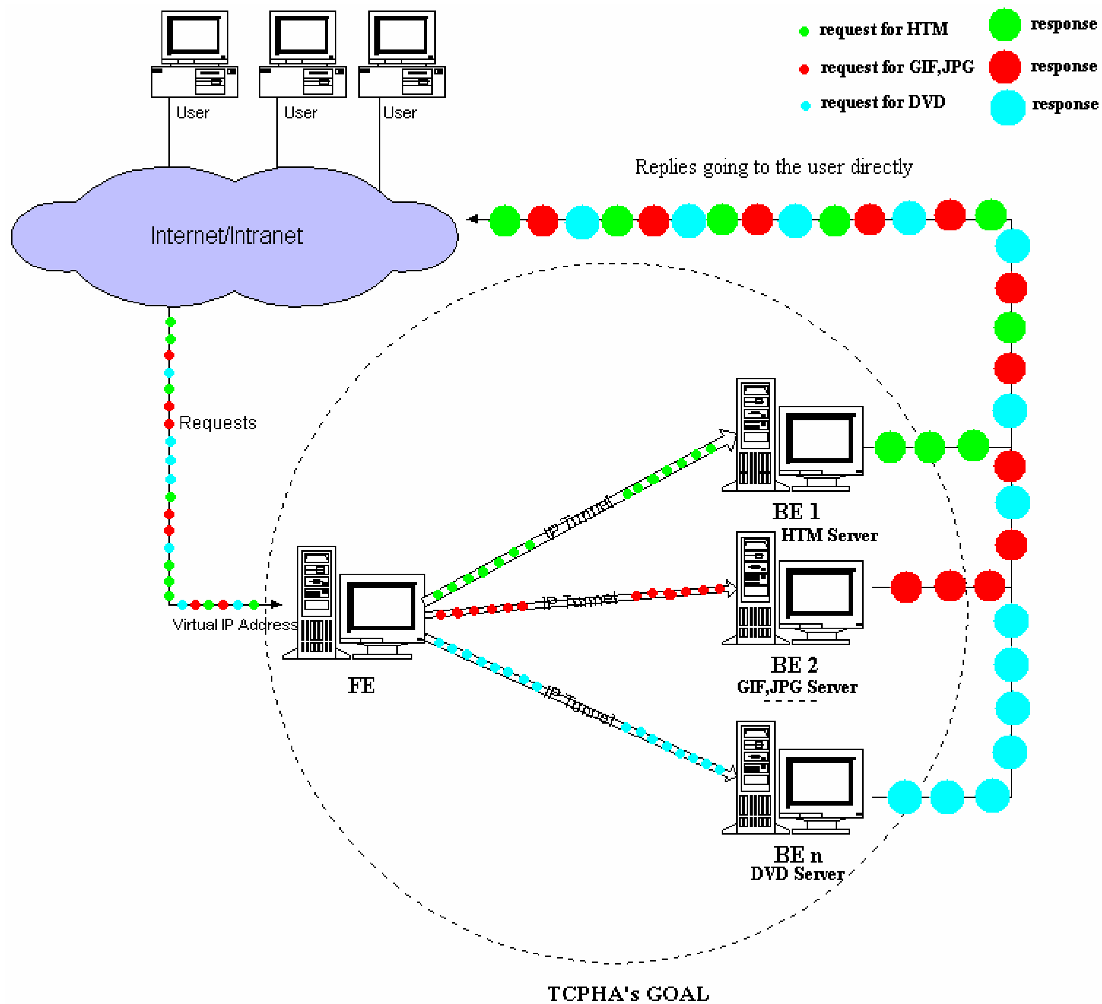

Figure 1. 1 TCPHA's Goal

It distributes the requests by content to BE, BE serves requests, and sends response directly to client. System efficiently avoids the FE bottleneck problem existing in popular server clusters, bring to system higher scalability. Furthermore, it bring BE high cache hit rate, which will greatly improve system performance. So TCPHA combines strong points of popular layer-4 and layer-7 schedule system, overcomes their shortcomings. It has high performance.

TCPHA is implemented based Linux 2.4.20 kernel, developed with C. It has two main releases: 0.2 release and 0.3 release. 0.2 release is stable, 0.3 release try to adopt a

novel technique to support P-HTTP: Multi Connection Handoff, it is under tests. TCPHA runs inside the OS kernel, implements TCP Handoff, ARP filtering, kernel symmetric multiple-thread transaction-driven architecture, dynamic IP tunnel, HTTP packet zero copy relay techniques etc. It efficiently avoids the overhead of context switching and memory copying between user-space and kernel-space. System is implemented as loadable kernel device driver module, no need modifying OS network stack. No need modifying user space server application codes and client browser codes, everything is transparent from user space perspective and from client perspective. System installation and configuration are very simple. System also support regular expression rule setting, user can thus set quite complicated schedule rules.

TCPHA has been published in internet (http://dragon.linux-vs.org/~dragonfly/), is attracting more attentions over the world, and is accepted by well known LVS project (Linux Virtual Server Project, http://www.linuxvirtualserver.org/) as subproject of it.

2 System Architecture

TCPHA system architecture is shown in figure 3.1:

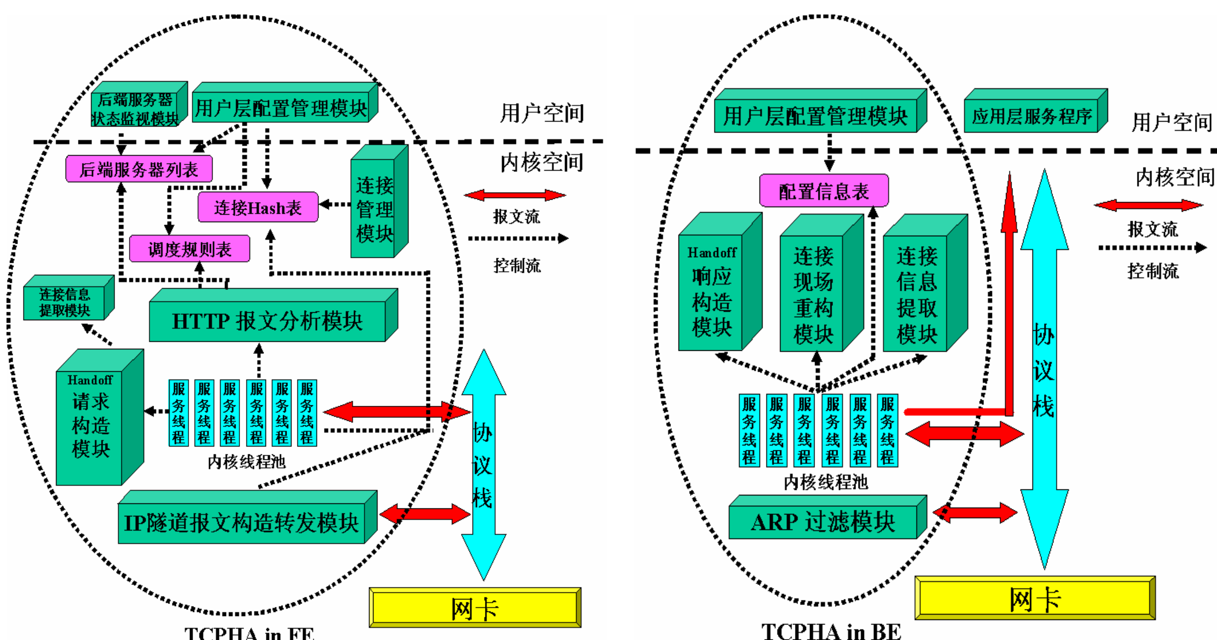

Figure 2. 1 TCPHA architecture

TCPHA is composed of tcpha_fe (dispatcher), tcpha_be (real server). It runs inside OS Kernel and is implemented as loadable device driver module. Its installation is very simple, no need making any modifications to OS kernel. More details about TCPHA architecture are as follows:

FE

- **Connection Management Module,** Manage the persistent connections with BE, maintain a persistent connection pool. When FE wants to send handoff request to one BE, it is responsible for assigning an idle persistent connection with this BE to FE.
- **HTTP Analysis Module,** Analyze HTTP requests according to HTTP protocol. Search schedule rule table by HTTP packet content and BE listing, choosing a BE.
- **Kernel Thread Pool,** Maintain server daemons. Once a client request is received, it assigns an idle server daemon to serve the request.
- **Handoff Request Constructing Module,** Construct handoff request according to handoff protocol. Call connection information extracting module to acquire connection information.
- **Connection Information Extracting Module,** Extract connection information, such as client address and port.
- **IP Tunnel Packet Constructing and Forwarding Module,** Intercept successive packets on migrated connections in IP layer, encapsulate packets according to IP Tunnel protocol, forward them to chosen BE.

BE

- **Handoff ACK Constructing Module,** Construct handoff ACK according to handoff protocol and handoff result, send out to FE.
- **Connection Reconstructing Module,** Reconstruct connection data structure, relay it to user space server application.
- **Connection Information Extracting Module,** Extract connection information from handoff request.
- **Kernel Thread Pool,** Maintain server daemons. Once a handoff request is received, assign an idle server daemon to serve it.
- **ARP Filtering Module,** Process ARP packets. Details about ARP problem see chapter 4.4.

The architecture of 0.3 release is roughly the same with 0.1 release, just more complicated. For 0.3 release is still under tests, here we won't introduce it more, details see chapter 4.7.

3 Working Principle

System (0.2 release) working principle is as follows: Client sends a TCP connection request to FE, TCPHA on FE assigns an idle server daemon to serve request from kernel thread pool, creates a connection with client. Client sends HTTP request. When server daemon receives the HTTP request, it calls HTTP analysis module, parses the request according to HTTP protocol, extracts information for schedule, such as URL. Then it searches schedule rule table, chooses a BE. Then searches BE listing to acquire details about this BE, such as IP address, port, load. Next it calls handoff

request constructing module to construct handoff request according to handoff protocol. Then it assigns an idle persistent connection with chosen BE from persistent connection pool, sends out the handoff request to chosen BE.

Server daemon on BE receives handoff request, first checks the 'magic number' field to confirm handoff packet. Then calls connection information extracting module to extract connection information from packet, modifies some fields in 'sk_buff' to let packet seems to be the original HTTP request from user space application perspective. Next it calls connection reconstructing module to reconstruct connection data structure. Last it calls handoff ACK constructing module to construct handoff ACK, sends it out to FE.

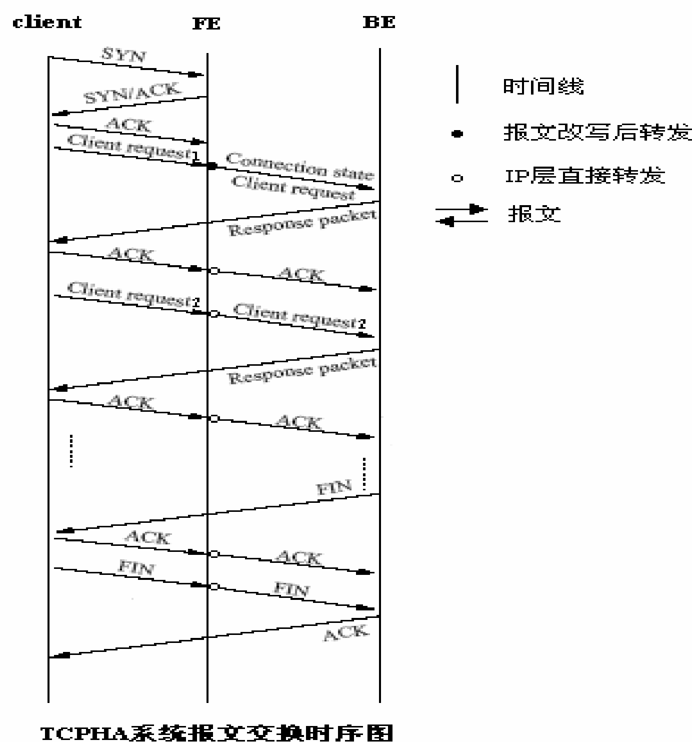

Figure 3. 1 TCPHA packets flow

TCPHA on FE receives handoff ACK, checks 'magic number' field and 'conn_magic' field, then extracts handoff operation result information from it. If handoff operation is successful, TCPHA will reset the connection. Note, here mentioned 'reset' is only clearing all the connection data structure, but doesn't initiate a normal four-way handshake course to close connection. It is the same behavior with receiving RST packet on the connection or connection timeout. Then TCPHA on FE will register the four-tuple of this connection and destination BE information to connection hash table which will be queried by IP tunnel packet constructing module. That module will intercept the successive packets on this connection and forwards them to chosen BE in IP layer.

TCPHA packets flow is shown in figure 4.1, First Client initiates a normal three-way handshaking to create a TCP connection with FE. Then client sends request 1, this packet is modified by FE, then is forwarded to BE. In BE, this connection is reconstructed. BE will send ACK directly to client, bypassing FE. The successive packets will be forwarded to BE in IP layer by FE. Responses will be sent directly to client by BE.

The system process of 0.3 release is roughly the same with 0.2 release, only to persistent connection, 0.3 release may initiate multi-handoff, details see chapter 4.7.

4 Key Techniques

4.1 Handoff Protocol

FE needs transmitting characteristic information of TCP connection and original HTTP request to scheduled BE, BE should send ACK to FE. So an application layer communication protocol is needed. we name it Handoff protocol. Its details are as follows:

Handoff Request, it contains connection information and HTTP request. In the headroom of HTTP request packet buffer, we inject a handoff request header between TCP header and HTTP header, its format is shown in figure 5.1. First 32 bits is TCP Handoff packet identifier, its value is 0x12968b9. Next 32 bits is connection identifier, its value is the next sequence number will be received on this connection. Next is a 'conn_info' structure, which contains connection information. After that is the original HTTP request packet. Handoff request is sent to BE by FE when FE initiates a TCP handoff.

Figure 4. 1 Handoff request

Handoff ACK, Handoff ACK packet format is shown in figure 5.2. 'Magic number' and 'conn_magic' are the same with Handoff request. 'Msg' is an enumeration variable. It indicates the handoff result.

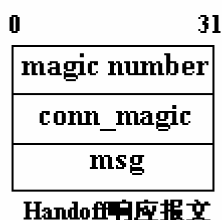

Figure 4. 2 Handoff ACK

The value of 'conn_magic' is copied from the corresponding handoff request. So FE

can confirm the ACK is to which request. Handoff ACK is sent to FE by BE after BE has finished the handoff operation.

As we have seen, only one packet exchange is needed during handoff course, so the performance is high.

4.2 TCP Handoff implementation

In almost all previous socket handoff systems, they adopt faked 3-way handshaking technique. The core idea of it is adding a module in the TCP layer in the network stack. It fakes the client, does the three-way handshaking with network stack, namely it generates the SYN, ACK packet the same with client sent to FE, sends them in turn to network stack. In fact, the technique performance is low, and is unnecessary. In TCPHA, we propose Agile Handoff whose concept is borrowed from agile software developing to implement TCP Handoff.

Agile handoff is shown in figure 5.3, primary modules are SHS (SH sender), PR (packet router) and SHR (SH Receiver). The working principle is as follows:

1. When FE decides to initiate a handoff, it informs SHS. SHS collects connection information, rewrites HTTP request, and adds the handoff request header.

2. SHS chooses an idle persistent connection which is beforehand created with scheduled BE from the connection pool, sends out the handoff request to SHR on scheduled BE.

3. SHR on scheduled BE receives the handoff request, extracts the connection information and reconstructs the connection data structure, and uses HTTP zero copy relay technique to queue the original HTTP request to the receive queue of the newly created connection, uses connection relay technique to relay the connection to the user space server application, which will serve the HTTP request, sends out the response directly to client.

4. SHR constructs handoff ACK, sends it to FE.

5. SHS on FE receives handoff ACK, destroys the data structure of the connection, informs PR the four-tuple of the connection and BE address, PR will forward the successive packets on the connection to BE in the IP layer.

As mentioned above, only five steps are needed in the TCP handoff course, only one packets exchange is needed between FE and BE, so the performance is high. And the entire course is transparent from client and user space application perspective. So no need making any modifications to client and user space application.

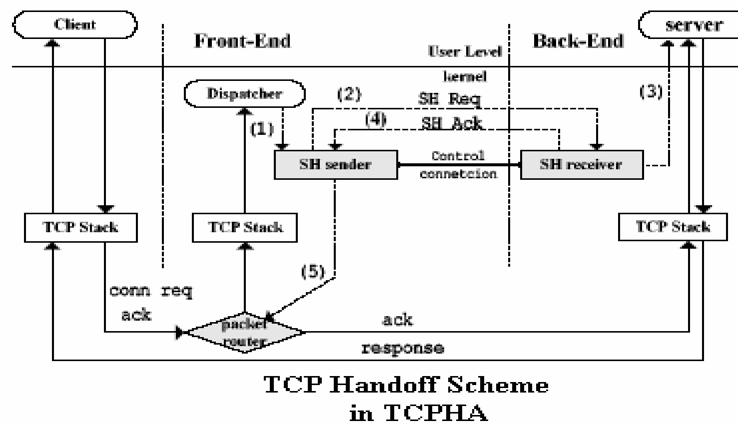

Figure 4. 3 TCP handoff implementation

Agile handoff constitutes of three sub techniques: connection reconstructing, connection relay and HTTP request zero copy relay.

4.2.1 Connection Reconstructing

Connection Reconstructing technique utilizes the connection information to reconstruct connection data structure inside BE runtime kernel. By studying the typical Web server programming model, we will discover that a typical Web server program is in a infinite loop: listening connection requests from clients on a listen socket (‘accept()’ system call), if receives any request, Operation system network stack does the three-way handshaking according to TCP protocol to create a TCP connection with client, then return a socket descriptor to application to identify the newly created connection (the return value of ‘accept()’). Application uses the identifier to transmit data with client, and then closes the connection. Certainly, programmer may adopt multi-process or multi-thread to parallelize waiting requests and serving requests.

As above analysis, TCP connection is created by ‘accept()’ system call, and the course is done by operation system network stack, which is transparent from user application perspective. User application operates the new connection only by the return value of ‘accept()’ system call, which edifies us that to reconstruct connection, we need only simulating accept() call, constructing connection data structure, registering in the system hash tables. Besides, it needs cooperating with FE to acquire connection information, such as remote IP address, port etc.

Steps are as follows:

(1) Allocate ‘sock’ structure and initialize it;
(2) Search the route to client, fill in the ‘sock’ structure with route information;
(3) Extract the connection information from handoff request;
(4) Modify the ‘sock’ structure according to connection information;
(5) Hash the ‘sock’ structure to system hash tables.

4.2.2 HTTP Request Zero Copy Relay

After connection reconstructing, the HTTP request is needed to queue to receive queue of the newly created connection, then using connection relay technique to relay the connection to user application.

In Linux, for each packet, before it is copied to user buffer, it is placed in a kernel buffer, traverses the network stack, and a ‘sk_buff’ structure is allocated to record control information. Each protocol layer operates the packet through the ‘sk_buff’ structure. As our system runs inside the OS kernel, can access kernel data structure, including the ‘sk_buff’ structure. So we can modify the ‘sk_buff’ structure, change the owner of the packet to new connection, and let packet seen from user application perspective to be the original HTTP request sent by client, leave the packet buffer untouched. So the HTTP request can be relayed to user application with zero copy.

Steps are as follows:
(1) Increase the value of ‘data’ field of ‘skb’, skip the handoff request header;
(2) Modify ‘copied_seq’ field, indicate the packet has been accepted by network stack;
(3) Drop the old route;
(4) Change the owner of the packet to new connection;
(5) Queue the packet to receive queue of the new connection.

4.2.3 Connection Relay

TCP connection is a static concept, it can be regarded as a resource, it has owner. The owner is process. Connection reconstructing only constructs the connection data structure, we must deliver it to owner, only that it can be utilized by owner. To this case, owner surely is the user space Web server application. How to let user space Web server application accept the fake connection, is a key technique. Here we explore connection relay, it can relay a connection from a process to another listen process.

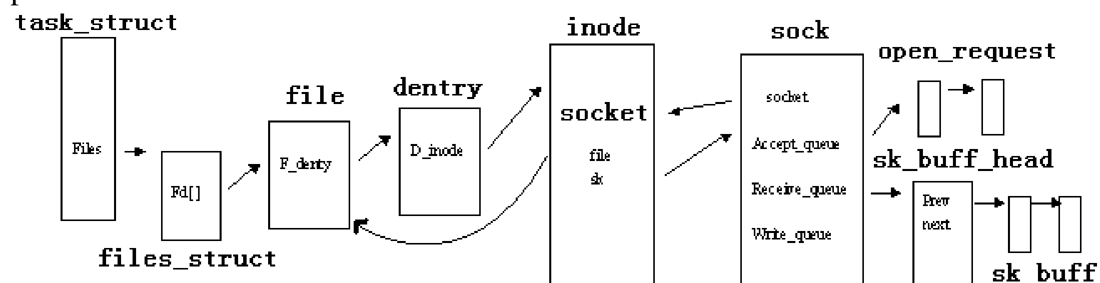

Figure 4. 4 Linux network data structure

As shown in figure 5.4, In Linux TCP/IP network stack implementation, there is an

important data structure 'sock', all the TCP connection information is recorded in this structure. TCP connection is 'one-to-one' with 'sock' structure. Structure 'sock' links to PCB (process control block) 'task_struct'though structure 'socket'. Receive queue contains all packets received on this connection, Write queue contains all packets will be sent. Accept queue contains all just-finished three-way handshaking TCP connections. Structure 'open_request' represents a just-established connection, the 'sk' field of 'open_request' points to connection structure 'sock'. Usually server application listens on listen socket, once client creates a connection with server, OS kernel will create the structure 'open_request' and structure 'sock', and queue it to accept queue, wake up the listen process, which leads to the 'accept()' system call return. Listen process can not distinguish that the connection is whether be created by OS kernel network stack, it as well doesn't concern that. It only thinks that client has created a connection with it.

Steps are as follows:

(1) Break the link between 'sock' structure and source process PCB;
(2) Fake a 'open_request" structure, fill in it;
(3) Link 'sock' to 'open_request';
(4) Link 'open_request' to 'accept_queue' of destination process PCB;
(5) Wake up destination process.

When finished connection relay, from user space application perspective, it can not distinguish that whether the connection is created by TCPHA or OS network stack, it only thinks that some client creates a connection with it, and sends a HTTP request. So it serves request, sends response directly to client.

4.3 Symmetric Multiple-Thread Transaction-Driven Architecture

The architecture of server application has important influence on server performance. To highly parallelize server, we exploit Symmetric Multiple-Thread Transaction-Driven Architecture in TCPHA, and adopt the kernel thread mechanism supplied by Linux kernel to implement TCPHA server efficiently.

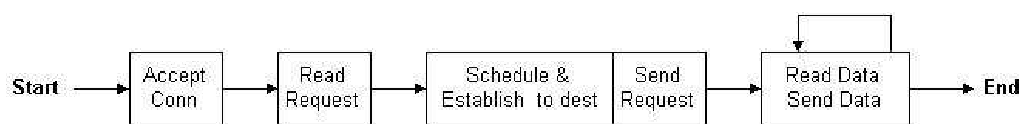

Figure 4. 5 TCPHA process flow

Steps are as follows:

- Accept connection: Accept client connection request by 'accept()' system call, create socket.
- Read request: Read request from socket.
- Schedule and connect to destination server: Parse request, assign a BE, assign an idle persistent connection from persistent connection pool. If no idle connections,

create a connection with BE.

- Migrate connection: Send connection information to BE, BE reconstructs connection by using connection information.
- Reset connection, enroll connection: When receives handoff ACK, reset the connection, record the four-tuple of the connection to hash table.

In TCPHA implementation, we adopt Linux kernel thread to implement Symmetric Multiple-Thread Transaction-Driven Architecture. Linux kernel thread is different with usual thread concept, it is intervenient between process and thread. Kernel threads share OS global variables, but their execution is independent. If one kernel thread suspend, other kernel threads won't suspend again. Furthermore, kernel thread runs in ring0 privilege, it directly accesses kernel memory, so the page swapping and system call overhead are avoided, so its performance is high. In transaction-driven implementation, we modify the 'poll()' system call in Linux kernel to permit kernel thread call it, which supply a novel programming model for kernel thread.

4.4 ARP problem and the solution: ARP Filtering

As said above, FE and BE share VIP address. When client want to connect to VIP, it will first send an ARP request to acquire MAC address of VIP. Our goal is making client create connection with FE, so we must let BE won't send out ARP response. Or else client will directly connect with BE, FE is bypassed. FE can not perform load balancing function. The cluster configuration is exposed to client, which may bring hidden trouble in security. The worse is that due to ARP caching, large numbers of clients will access the same BE at the same time, other BE are idle, which leads to load unbalance. Or when client is transmitting data with one BE, it receives another ARP response from other BE, it changes the ARP table entry. Then the data is sent to another BE, surely it will receive a RST from there. And if the resource is classified to store by content, the data set on different BE is different. If client create connection directly with BE, it may not acquire the request content. So ARP problem is a key problem.

Before Linux 2.0, solving this problem is very simple, for some devices (named 'nonarp' device) won't do ARP response, such as tunnel0, dummy0, lo0. So we only need configuring VIP on these devices. But from Linux 2.2, all these devices will do ARP response. Current systems, such as IPVS, adopt a kernel patch to solve it. Here we utilize the Netfilter framework supplied from Linux 2.4, propose another solution, we name it ARP Filtering. TCPHA hooks the ARP_IN chain and ARP_OUT chain, to intercept/rewrite ARP packets. When an ARP packet arrives, it will be sent to ARP_IN chain and intercepted by TCPHA. If it is ARP request and destination is VIP, TCPHA drops it. If the source address is VIP (sent by FE), here need some special operation.
For BE will think that the packet is sent by itself, it will directly drop it in network stack. Then FE won't know BE's existence, it as well can not forward packets to it. At the same time, we must let BE don't know that there are some hosts address is the

same with itself, which may incur some unknown troubles (for example, Windows will inform user IP address conflict, disable the IP address), so we modify the source address to an un-existing address X. When an ARP packet will be sent, it will be sent to ARP_OUT chain and intercepted by TCPHA. If the source address is VIP, TCPHA modify it to IP tunnel device address, if destination is X, modify it to VIP. By such operations it can not only avoid BE inform its VIP, but also let FE know the BE's existing. In fact, VIP on BE is a silent IP address, only known by itself. And for TCPHA is implemented as a loadable device driver module, after it is unloaded, ARP filtering will abolish, which won't influence normal network stack behavior.

4.5 Regular Expression Rule Matching

As we all know, regular expression is a powerful tool for pattern matching and replacing. We can find regular expression in almost all UNIX based tools, such as vi editor, Perl or PHP script, and awk or sed shell program. User can set very complicated pattern by using regular expression. In TCPHA, we supply regular expression rule matching support, so administrator can set complicated schedule rule to load balancing. For example, storing image files and HTML files on different BE, setting file extend name matching rule to load balancing. In implementation, we only port regular expression library written by Henry Spencer to kernel.

4.6 Dynamic IP Tunneling

IP tunneling technique encapsulates an IP packet into another IP packet, which makes the packet can be forwarded to another destination. IP tunneling is also named IP encapsulation. IP tunneling is mainly used in mobile host and VPN (Virtual Private Network), in which the tunneling is created statically, the IP address of two endpoints is fixed and exclusive.

We adopt IP tunneling to forward packets to BE, it is because that if forward directly, the packet destination is FE itself, the packet won't be sent to network physically. Furthermore, there are many BE not one, so we can not create tunnel statically, but choose a BE dynamically, encapsulate the packet according to IP tunnel protocol and sends it out to BE. Here we studied the Linux source of IP tunnel protocol, made reference to it to implement dynamic IP tunneling technique.

Steps are as follows:

(1) Search the route to BE;
(2) Extend the headroom of packet, leave a IP header room;
(3) Fill in the IP tunnel protocol header;
(4) Calculate checksum;
(5) Send it out to network.

4.7 Local Node Feature

If the local node feature is enabled, FE can not only redirect the packets of the specified port to BE to process it, but also can process the packets locally (local node). Which node is chosen depends on the scheduling algorithms. This local node feature can be used to build a virtual server of a few nodes, such as 2, 3 or 4 nodes, in which it is a resource waste if FE is only used to redirect packets. It is wise to direct some packets to the local node to process.

This feature can also be used to build distributed identical servers, in which one is too busy to handle requests locally, and then it can seamlessly forward requests to other servers to process them.

4.8 P-HTTP Support (Under Discussion)

HTTP/1.1 introduces persistent connection support by ‘keepalive’ option. In TCPHA 0.2 release, it will forward successive requests on migrated connection to the first scheduled BE, in other words, TCP handoff is only done once. It is efficient enough to software download website, media service website etc. These websites will return a great deal of data when receive a request. In TCPHA 0.3 release, we propose multi-handoff technique to support P-HTTP more efficiently. The core idea is: To persistent connection, let BE take part in scheduling too, which can solve FE performance bottleneck problem efficiently. FE only schedules the first request on the persistent connection, and migrates the connection to chosen BE. If successive request arrives, it is up to the scheduled BE to schedule again. Here BE’ function is the same with FE, it parses new request, does scheduling according to rules. If it schedules to itself, handle it directly. If it schedules to another BE, it initiates a handoff course, migrates the connection to new BE, resets connection, informs FE new BE address.
This needs BE knowing the configuration of cluster. If connection data structure has multi copies in different BE, the state synchronization is very complicated, and some unpredictable influence may occur on client. Such as one BE closes the connection when another BE is transmitting data on this connection. So the key point of multi-handoff is making only one connection data structure existing in BE.

4.8.1 Single Handoff Course

1. Client C creates a TCP persistent connection with kernel thread T on FE, T creates corresponding data structure. Client sends the first HTTP request to FE, T receives it, sets the connection status HANDOFFING, and records the connection in hash table. For System hooks the NF_IP_LOCAL_IN and NF_IP_FORWARD chains, program in BH (Bottom Half) layer (abbreviate to BH, the same below) can handle the successive packets on the connection according to hash table. To HANDOFFING

status, BH will drop all successive packets on the connection to make the handoff course proceed atomically. T parses request, does scheduling, assumes it schedules to BE_1, T acquires connection information from the connection ‘sock’ structure inside runtime kernel, constructs handoff request and sends it out to BE_1.

2. BE_1 receives handoff request, reconstructs connection, relays the connection to user space HTTP server, and sends out ACK to FE. Kernel thread T on FE receives ACK, records the BE_1 address in the hash table, and sets the connection status HANDOFFED. To this status, BH will forward all input packets on this connection to BE_1, drop output packets. Then FE resets connection.

3. Once FE watches the FIN or RST packet on this migrated connection, it sets the connection status TIME_WAIT. After some time, it clears the connection information from hash table.

Fault Tolerance:

1. In handoff course, if the packet FE sent to BE_1 is lost or BE_1 is not available, FE won’t receive ACK from BE. When timeout, FE closes the connection with client, clears the connection data structure in hash table.

2. If response is lost, handle method is the same with above.

4.8.2 Multi-Handoff Course

The prophasic work process is the same with single handoff course. Besides, BE records the migrated to it connections and takes part in scheduling too. System hooks NF_IP_LOCAL_IN and NF_IP_FORWARD chains, to facilitate BH handling the successive packets on the connection though hash table. First BE_1 sets the connection status ESTABLISHED. To this status, BH algorithm: Receive all TCP control packets (zero payload packets). If set FIN or RST in packets, modify the connection status to TIME_WAIT. If payload is not zero, regard it as a new HTTP request. Here BE_1 does the same jobs with FE. BE_1 pareses the request, does scheduling. If BE_1 schedules to itself, it receives the packets directly. If BE_1 schedules to BE_2, BH on BE_1 delivers the packet to kernel thread T’ on BE_1, sets the connection status WILLHANDOFF, return NF_STOLEN. To WILLHANDOFF status, BH receives all TCP control packets, If FIN or RST is set in packet, set the connection status TIME_WAIT. If payload is not zero, drops it. It is to ensure that before new handoff course has been finished, BE_1 won’t receive new requests. T’ receives the HTTP request, connects to BE_2, searches the connection ‘sock’ structure in network protocol, if the ‘sock’ structure indicates that the connection status is TCP_ESTABLISHED, and hash table indicates that connection status is WILLHANDOFF, then T’ changes connection status to HANDOFFING. To this status, BH will drop all input and output packets on this connection. BE_1 extracts the connection information, constructs handoff request and

sends it out to BE_2. BE_2 does the connection reconstructing, relays the connection to user space HTTP server, and informs BE_1 that handoff is successful. When BE_1 receives ACK, resets the connection, sets the connection status HANDOFFED. To this status, BH on BE_1 will forward all input packets to BE_2, Output packets return NF_STOLEN. When timeout, BH will delete the connection from hash table. Then T' on BE_1 will inform FE forward destination changes to BE_2, When receives FE ACK, BE_1 will delete the connection from hash table.

Algorithm key: To make connection safely migrate between BE, the key is to ensure that only when BE_1 has received ACK to HTTP response then does connection handoff. For example, client requests index.htm, FE migrates the connection to BE_1, BE_1 serves request, sends out the file to client. Client receives it, sends out ACK. If before BE_1 receives ACK, BE_1 migrate the connection endpoint on BE_1 to BE_2 (according to HTTP/1.1, it is possible, such as client sends two requests at the same time), then BE_2 will find that need retransmitting index.htm, but if the file is classified to store by file type, BE_2 will have no such file, say nothing of the retransmit buffer pointer handling problem. In fact, the key is to let BE_2 see a new request need to be served, not retransmitting a response. Surely this problem can be solved by totally reconstructing connection including retransmitting buffer too. In another case, if the BE_1's response is lost, client will retransmit the request for index.htm, this won't be a problem, for this will make BE_1 schedule to itself, BE_1 will receives the request and serves it again.

In this algorithm, we add the WILLHANDOFF status on BE. For from BH delivers the packet to kernel thread T' to T' determines to do handoff, it must has some delay. In this time, the connection may be closed, so we need checking connection status in 'sock' and hash table before doing handoff. Another measure, dropping all new requests, but receives all control packets, is also to ensure that BE_1 receives ACK before doing handoff. Also setting HANDOFFED status is to let ACK be sent to BE_1. When BE_1 receives ACK from FE, it deletes the connection entry from hash table immediately. It is because that BE_2 may migrate the connection back to BE_1, the connection entry in hash table on BE_1 may influence the connection handling.

4.8.3 BE Scheduling Technique

BE scheduling technique is the key to implement multi-handoff. Let BE take part in scheduling, compares with FE scheduling, has a difficult problem: When FE does scheduling, the connection has been created with client, HTTP request can be received directly by kernel thread on FE. But when BE does scheduling, the connection has been relayed to user space server daemon, then the new requests on this connection will be sent directly to user space server, kernel thread on BE can't receive it. In other words, the owner of this connection is user space web server, not system kernel thread on BE. Here we utilize Netfilter framework to solve this problem.

In BH layer, system intercepts successive packets on this connection, judges it is whether be HTTP request (If TCP payload is not zero, regard as it is). If not, sends it to network protocol stack to do normal process. If is, parses the packets. Note, for it is in IP layer, needs parsing TCP packet. Then parses HTTP, matches schedule rules. If schedules to another BE, takes over this packet, not send it back to network stack. Records the BE address and port in this packet, queues it to an asynchronous packet queue, sends signal to a particular kernel thread, it is up to that kernel thread to handle it farther.

Steps are as follows:

（1）Check validity of packet, For system runs in IP layer, checking procedures needed are:

i. Packet is whether be sent to local host, if no, deliver it to network stack;
ii. Packet is whether be sent to loopback device, if so, deliver it to network stack;
iii. Upper protocol is whether be TCP, if no, deliver it to network stack;
iv. Packet integrality check.

（2）Search hash table according to remote address and port in IP header, Acquire the connection data structure;

（3）Acquire connection status;

（4）Handle by connection status:

- ESTABLISHED:
  I. If FIN or RST bit set, set connection status TIME_WAIT;
  II. If TCP control packet (zero TCP payload), deliver it to network stack;
  III. Otherwise regard as HTTP request, do checksum check, scheduling. If schedule to itself, deliver to network stack; if no, record the new scheduled BE address and port in the packet, queue the packet to an asynchronous queue, send signal to kernel thread.
- WILL_HANDOFF:
  I. If FIN or RST bit set, set connection status TIME_WAIT;
  II. If TCP payload not zero, regard as new HTTP request, drop it;
  III. Otherwise deliver it to network stack.
- HANDOFFING: Drop it;
- HANDOFFED: Encapsulate it and forward it to newly scheduled BE.

（5）If kernel thread is notified, it checks the asynchronous packet queue, if finds it not empty, peeks an entry from it, it is a HTTP request. It extracts new BE address and port from it, creates connection with the BE, extracts the original HTTP request, acquires the connection information from network stack, constructs a handoff request, sends it out to newly scheduled BE.

（6）Get ACK from newly scheduled BE, if handoff is successful, inform FE the new BE address and port, reset the connection, delete entry from hash table.

4.9 High Availability

TCPHA supplies high availability for cluster. FE will send detection packets to BE periodically (the interval is settable, value 0 will shut down the function), BE will return response. In the next TCPHA release, we will include load information in response, implement dynamic load feedback. In current release, BE returns only work status. If in some time, FE doesn't receive response from BE, it thinks that the BE is not available. FE will mark the BE's status not available. When do scheduling, FE won't schedule to this BE. On the other hand, if BE comes back to available, it will respond to detection again. When FE receives response again, it will mark the BE's status available and take part in scheduling. So BE collapse won't influence cluster service.

4.10 Dynamic Scalability

TCPHA permits adding new BE to cluster in runtime, no need stopping or restarting cluster service. When a new BE is added, it will read the configuration file, acquire FE address and own schedule rule, send out register request to FE, inform its schedule rule. FE will process register request, add BE information to BE listing, send out register success response to BE. Then BE can take part in scheduling like other statically configured BE.

4.11 Journaling

TCPHA supplies powerful journaling function. User can choose recording the information in file: /var/log/messages or the file user specify, also can control the detail level. Most detailed message includes all request content and process course. User can monitor and debug the cluster system easily.