

An Approach for Network Forwarding Systems Quality

GEORGE G. MITCHELL

WILLIAM M. FITZGERALD

JAMES DOODY[†]

Department of Computer Science,
National University of Ireland, Maynooth.
Maynooth,
Co. Kildare.
Ireland.

[†]Department of Computing
Institute of Technology, Tallaght.
Tallaght,
Dublin 24,
Ireland.

{georgem, williamf} @cs.may.ie

Jimmy.Doody@it-tallaght.ie

Abstract

We present a design pattern for improving the reliability of forwarding failures within protocol processes and kernel routers. Our design pattern makes use of a localised Invisible Recovery (IR) process and a technique for optimising the Kernel process. We evaluate the current techniques that provide recovery within network systems; by doing so we then pinpoint the different approaches and the benefits of our IR technique has over others.

We propose to optimise the Kernel process so as to increase the QoS of a software router. Our IR process makes use of an envelope surrounding an unchanged software protocol layer and operates in a Super Server fashion modelled on the Inetd daemon common in LINUX systems. This envelope stores just sufficient information on each connection to enable an invisible recovery of the protocol layer after a software fault. IR provides the means to permit a reconnection to take the place of the original connections.

1 Introduction

Modern software design methodologies are advancing yearly, the reliability of software has been improved, unfortunately experience has shown that the use of these methods alone can be inadequate. With modern Safety Critical Systems such as those in Avionics Control and those that must integrate with an existing legacy system, the use of high availability software has become necessary.

Thorough *optimisation* of existing systems and also the introduction of *software fault tolerance* it has been possible to provide this improved quality of service. One important thing to note about the implementation of fault tolerant systems is that they generally may be implemented in two differing ways, the most

predominant technique currently in use is that of code invasive *design time* fault tolerance. The second and also the far less expensive technique is software fault tolerance through *non-invasive coding*. The fault tolerant technique, which has been used in experiment 2 of this project, has been a non-invasive coding type. This has allowed us to extensively reuse existing software, for example the UDP protocol prototype.

It is notable that the least code invasive mechanism clearly reduces the potential for the number of possible faults that could be injected in the system. It is from this stance that the use of IR betters similar techniques for network fault tolerance.

The provision of improved routing in operating systems has a connection with Invisible Recovery that being by improving the underlying network communications the perceived quality of the network is greatly improved. Experiment 1 is still in preliminary stages, through kernel tracing and also measurement of packet routing times it will be possible to optimise the kernel of operating systems. This will enhance the quality of service provided to users.

1.1 *Software Reuse*

Reliability of software is one of the four metrics that reuse is measured on for its success or failure, therefore it is not surprising to find that reuse has found a place in the development of highly available systems. Through the standard development of fault tolerant systems the introduction of untried and untested methods can lead to the deployment of a system prone to errors. Fault tolerant systems have a simple paradigm *add only what you need to add*, code invasiveness is also required to be kept to a minimum. It is interesting to note that communications RFC's and research papers [1] in the area of fault tolerance are directed at developing new protocols that provide reliability to the protocol through code invasive techniques rather than stand alone non code invasive techniques as we discussed in Mitchell [2].

Both of the experiments in this paper can be regarded as primarily software reuse applications. The reuse of the existing Linux kernel and the reuse of network protocols has provided a sturdy test bed onto which can be built a generic technique for a thoroughly improved QoS for network communications systems.

2 **Related Research**

The related research to this paper clearly divides into two fields those dealing with routing and those dealing with fault tolerant network processes.

A number of tools have been developed for tracing the performance of the kernel [3]. FKT requires the introduction of probes in to existing code this has both advantages and disadvantages. The main advantage is that the user has the ability to insert any number of probes and has discretion as to where these are placed. Disadvantages include the requirement to recompile the kernel following the insertion of a new probe and also the possibility of the introduction of software errors. FKT is an advancement on previous kernel traces such as KernInst [4] and also Kitrace [5] both of which are runtime trace techniques, where by traces are introduced during the operation of the kernel.

Significant research has been conducted in the area of fault tolerance applied to processes and also protocols in message logging and checkpointing techniques such

as those developed and used by D.B. Johnson [6] and Elnozahy [7]. Using the well-developed checkpointing technique together with a logging system allows for both forward and backward recovery, this enables the systems to provide a high degree of reliability to the communication system. Johnson designed two main techniques *pessimistic* and also *optimistic* message logging.

For pessimistic message logging a new sender-based message logging protocol was developed. Each message was logged to a local volatile memory on the sending machine and ordering of the received messages was organised by a receive sequence number. Logging of messages overlapped the execution of the receiver until the receiver attempted a new send message. With this form of fault tolerance applied to a communicating protocol system applications had an overhead of just under 16 percent and an average overhead that measured 2 percent or less depending on the size and message volume.

Optimistic message logging outperformed pessimistic logging since the logging occurred asynchronously. Johnson presented a new optimistic message logging system. This guaranteed to find the maximum possible recoverable system state which had not been previously attained by other optimistic methods. All messages *and* checkpoints are utilised in his method and thus some messages received by a process before checkpointing were not necessarily required to be logged. With this technique overhead was kept to between 1 and 4 percent

3 Experiment 1

By measuring the performance of Linux routers we are able to obtain information which will be used in the final stage of this project. The final stage being the optimisation of the software router.

To measure the performance of a software router a trivial computer network was constructed. This consisted of a number of Linux PC's formed into a network as depicted in figure 1.

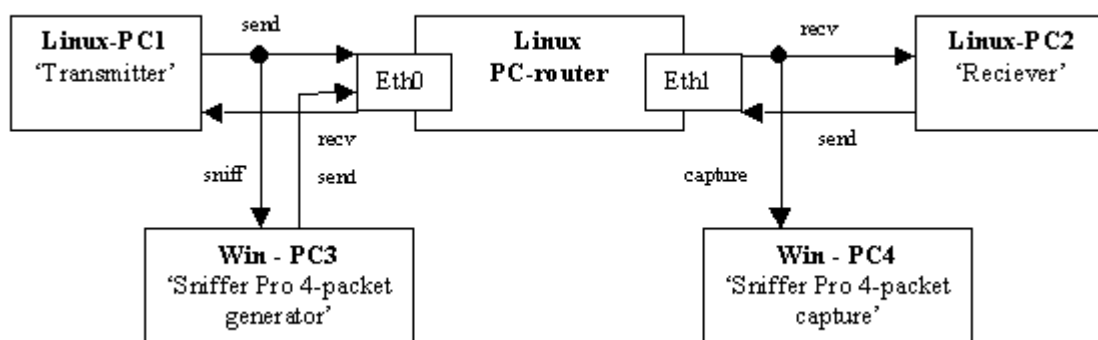


Figure 1 Software routing model

In this network the router is a PC that has been adapted, with the addition of a number of Ethernet cards and with an improved Linux kernel version 2.4.0-test9. This kernel includes the FKT tracing technique [3]. Linux-PC1 transmits a packet destined for Linux-PC2. During the transmission of this packet Win-PC3 sniffs the packets current state information, for example, the medium access control address, MAC.

Win-PC3 generates packet traffic using Sniffer Pro 4 [8]. This floods the router and we use Win-PC4 to capture the packets for comparison and further analysis.

Following the construction of this network, stress testing of the network has been conducted to determine weaknesses within the network. To-date the software routing systems has performed perfectly. It is our intention to stress the system to failure and following this to optimise the failed areas.

Following a rigorous examination of the underlying kernel, the network IP stack in regards to forwarded packets was performed to determine the “flow of control” throughout the system, this has provided a number of interesting points, we will now examine each of these in turn.

3.1 *The initial receive interrupt*

When a packet arrives at the network interface card, *NIC*, with an Ethernet frame, which matches the local MAC address, an interrupt is triggered. The network interface driver (*3c59x.c*) for this particular card handles the interrupt and fetches the packet data via DMA into RAM.

On further examination of the driver [9] we determined that, *vortex_interrupt()* initialised the *vortex_rx()* function which in turn allocated a *sk_buff* and also initiated the *netif_rx(skb)* to en-queue the newly formed *sk_buff*s.

Should a *sk_buff* not be time stamped by the driver, the *netif_rx(skb)* function places the time stamp. Following this, the *sk_buff* gets en-queued in the backlog queue for the processor handling this packet. Size is limited in this queue and should overflow occur packets will be lost. After en-queuing a *sk_buff*, the receive software interrupt is triggered for execution via the function *__cpu_raise_softirq()*. At this point the *vortex_interrupt()* handler exits and all interrupts are re-enabled.

3.2 *Receive Software IRQ's*

Within the older kernels from 2.2 down, Linux had adopted the idea of bottom half handlers. This mechanism defers execution of kernel tasks so that the interrupt CPU time is reduced to a minimum [10]. The bottom half handler: *net_bh()* which initially was part of *dev.c*, processed network packets in the wait transmission queue before handling the backlog queue. Within the Linux 2.4.0 kernel series the network stack has been converted to software IRQ, *softirq*, providing solutions to problems such as:

- Inability to handle more than 32 bottom half handlers.

- Bottom half handlers could only run on one CPU at a time.

Opening a *softirq* is accomplished through the use of the following:

```
open_softirq(NET_TX_SOFTIRQ, net_tx_action, NULL);
open_softirq(NET_RX_SOFTIRQ, net_rx_action, NULL);
```

Following this, a packet is managed in the network receive *softirq* (*NET_RX_SOFTIRQ*) which is called from the function: *do_softirq()*. At this point *net_rx_action()* is executed so that the networks *sk_buff*'s can be de-queued from the backlog queue using the lines:

```
local_irq_disable();
skb = __skb_dequeue(&queue->input_pkt_queue);
local_irq_enable();
```

3.3 Transmission of a packet via the IPv4 packet mechanism

At this point the main IP receive function: *ip_rcv()* is called from within *ip_input.c*. It verifies the packet header for correctness and handling routines for IP options.

Next a function call is made to *ip_rcv_finish()*. This initialises the virtual path cache for the packet. It describes how the packet travels inside the Linux network [9]. It makes reference to *ip_route_input()* from *route.c* which then calls *ip_route_input_slow()* from within the same program to make a decision on the packets next step.

On further examination it was found that if the packet is destined for the local host, then *ip_local_deliver()* is called otherwise *ip_forward()* is executed.

So if the decision was made to forward the packet using the *ip_forward()* function, a number of tasks have to be carried out. Firstly it must be discovered if the packets 'time to live' has expired, if so then reply an ICMP control message. Having picked a route we can now send the frame out after asking the firewall permission to do so. An ICMP HOST REDIRECT is generated giving the route just calculated. [9] If the packet needs to be fragmented then a call is made to *ip_fragment()*. After this a call is made to a netfilter hook:

```
NF_HOOK(PF_INET,NF_IP_FORWARD,skb,skb->dev,dev2,  
ip_forward_finish);
```

If the netfilter is returning a *NF_ACCEPT*, the function: *ip_forward_finish()* is executed. This checks to see if the packet needs any additional IP header options:

```
ip_forward_options(skb);
```

It later calls on *ip_send()*. When this has completed, at some point *ip_finish2_output()* is initiated to add the hardware header to our *sk_buff* before sending it onto the device driver, which passes the packet onto the *NIC* of another Ethernet. Note: Just before a packet is handed to the device driver, Linux traffic control takes place, see figure 2. Traffic control can decide if packets for transmission are queued or dropped [11,12]

Linux kernel-2.4.0 supports several flavours of queuing discipline which are Class Based Queuing, 'CBQ', Clark-Shenker-Zhang scheduler, 'CSZ', Random Early Detection queue, 'RED', Stochastic Fairness Queuing discipline, 'SFQ', Token Bucket Filter queue, 'TBF', First-In-First-Out, 'FIFO', and Priority queuing etc.

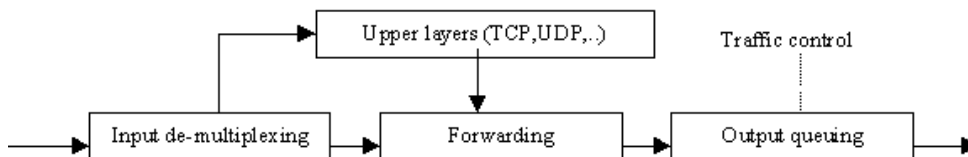


Figure 2 Processing network data

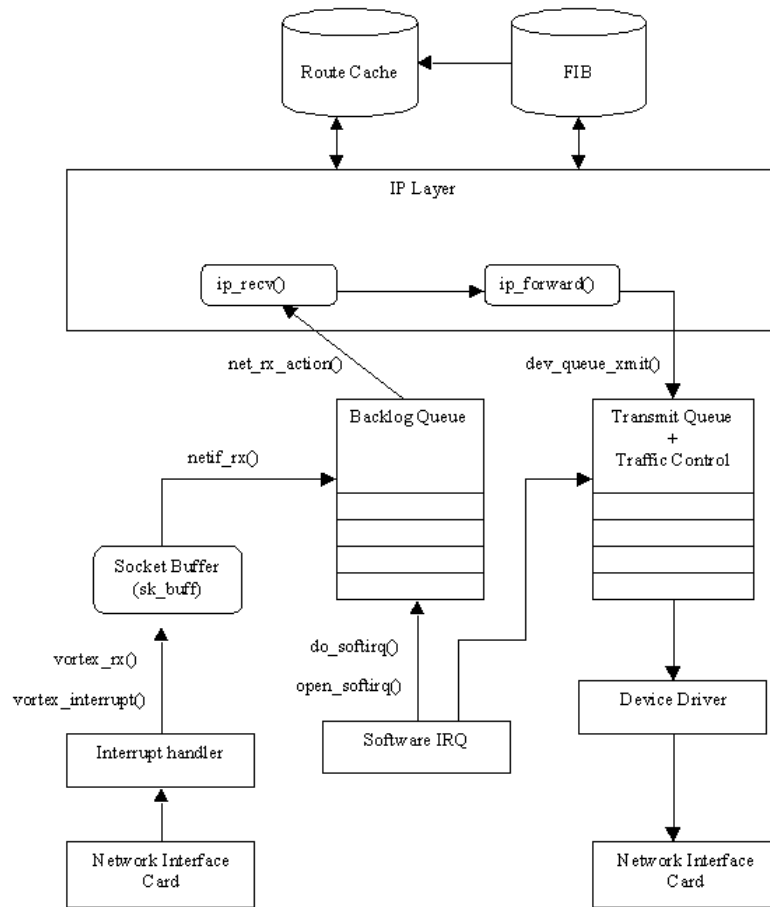


Figure 3 Model of Network Forwarding IP Stack

4 Experiment 2

Invisible Recovery, was designed to be platform independent and also to be applicable to most network protocols [13]. For communications protocols, irrespective of the connection orientation, it is necessary to mask the identity of the *Invisible Recovery* process along with its location. This masking is necessary so as to prevent a dependency which arises between any communicating process, port number and IP address being the most common. *Invisible Recovery* accomplishes this by being responsible for all out going requests from clients to servers similar in manner to the inetd super server that handles ftp requests [14]. Once *Invisible Recovery* is running it issues a system exec call for the desired server process. By doing so in a proxy server manner it is possible to hide the server behind the *Invisible Recovery process*. This method enables *Invisible Recovery* to monitor any number of clients and servers by maintaining an active database of incoming requests for service, see figure 4 and figure 5. Besides this envelope technique for hiding restarted processes, invisible recovery primarily consists of three parts:

1. Timing
2. Restart
3. Protocol Mangling

4.1 *Time*

Crucial to the correct operation of many network systems is the timing both of packets and the overall session. The OSI protocol stack introduced session layers that provided a compensation operation for intermittent connections by token management and check-pointing data transfer. This permits multiple network sessions to be integrated together to form one continuous data transfer. This type of method is not available in all network and telecom protocols for example, TCP / IP, the timing and also the non-interrupted communication between networked machines is necessary for the continuous operation of the protocols. By providing a proxy server *IR* to the client we maintain the acknowledgement of data packets and also enable the system to disguise the fact of restarting other servers, all of this is accomplished in under a couple of seconds when the entire system is in a non congested state, to understand the actual operation of *IR* it is necessary to examine its two fundamental parts; Restart & Protocol Mangling.

4.2 *Restart*

The initial communication required for a connection between networked computers normally requires the issuing of a number of protocol data units, *PDU*. Each of these are used to enable the client to register itself with the server, typically 3 to 5 *PDU*'s are required, in the TCP protocol, 3 *PDU*'s are used in the well know 3-way hand shake. Our *IR* system operates in the form of an envelope that surrounds an unchanged protocol entity, with this design it is possible to operate between upper and lower network layers providing a "restartable" component for most telecommunications and network protocols. Our envelope makes a data copy of all data that passes through the layer. On detection of a fault, our restart component reinitialises the connection by reissuing *PDU*'s and then creating a new connection. Our envelope, running as a process, uses a database containing the state information of the application process and also the protocol process. Typically this consists of; port number, IP address, unacked_data, last_seq_number, etc. This provides sufficient information to enable the *IR* process to restart and to then hide the failure of the underlying network entity, we accomplish this with the aid of *protocol mangling*.

4.3 *Protocol mangling*

This is essentially a mapping of all the last known good state information to the current new state information. It is necessary so as to cope with the newly regenerated protocol entity that will have been "reincarnated" with differing sequence numbering, possibly different port information and possibly incorrectly matched data. We use the stored data to correct the data problems of acknowledged verses unacknowledged data. Then by mangling the state information such as old_port address to new_port addresses, old_ip to new_ip and so forth we make it is possible to disguise the fact of a protocol entity failure, and continue to operate correctly.

4.4 *Invisible Recovery V's Checkpointing*

In order to appreciate the key benefits of the Invisible Recovery fault tolerant network technique, it is best that the basic operation of both checkpointed and *IR* fault tolerance techniques be highlighted.

Figure 6(A) depicts a simple client server model, the start S (client), the destination D (server), a Fault Tolerant Unit and a store that contains checkpointed and logged information. Important about this model is that messages M, travelling along channels C, and passing through the FT Unit are logged to stable storage. This stable storage forms the checkpointed data and non-deterministic information store. This stored data is reused to regenerate lost information in the event of an application software fault. This technique has one significant drawback, the size of the store is infinitely large and this has memory constraints on the continuous operation of the entire system.

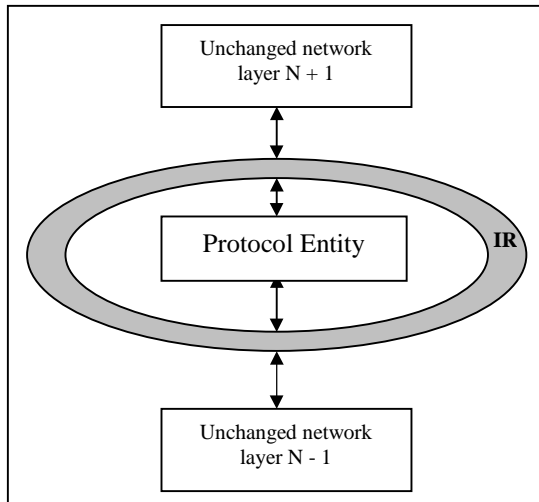


Figure 4 IR envelope communication

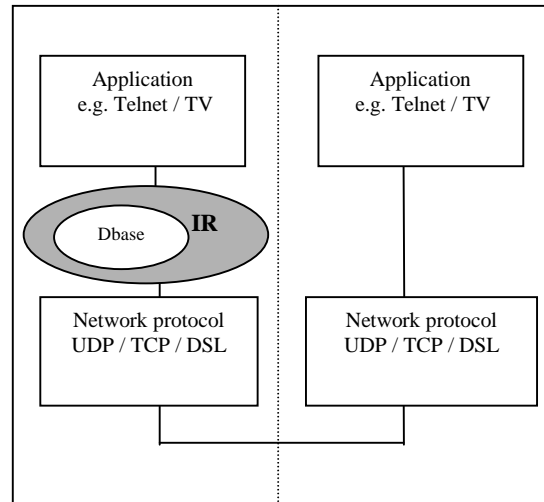


Figure 5 Visualisation of IR in Network

Examining figure 6(B) we see the basic operation of the Invisible Recovery technique, when comparing this diagram with figure 6(A) we see the addition of another store. This store contains the state information of the current correctly running process. On a software fault and failure of the process this information is used to mask the identity of the restarted process. The message store is required to store only incoming messages during a process failure and restart following this they are deleted and store remains empty until another failure occurs. The size of this message store is fixed with regard to the window size of the communications protocol (W) times the number of communications channels.

This limited size stores is one of the benefits of Invisible Recovery, what is also important is that these stores are outside of the control of kernel memory and this can reduce the chance of corruption of the entire system.

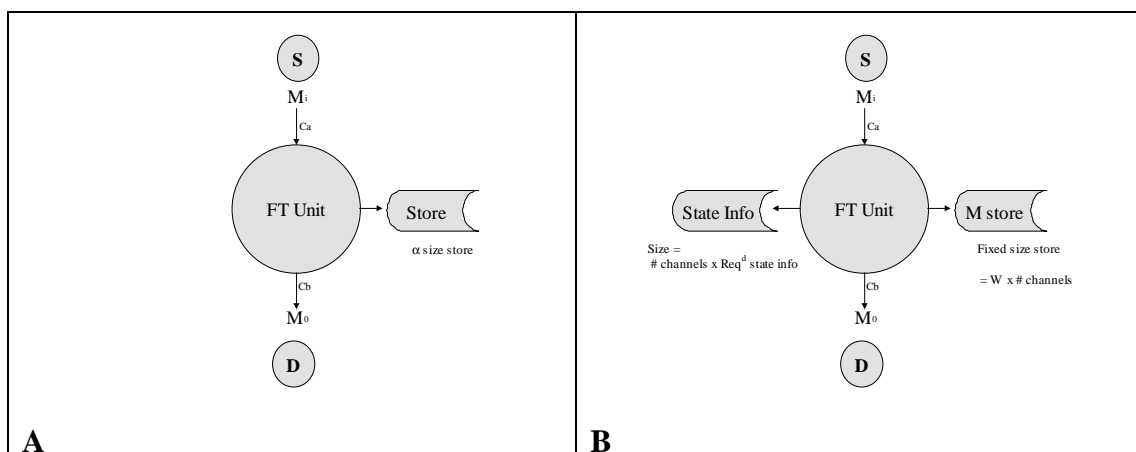


Figure 6 Checkpointed (A)& Invisible Recovery (B), Fault tolerant system.

5 Conclusion & Future work

In conclusion, see figure 3 regarding forwarding, our work in the future can now develop into examining the areas most likely to suffer packet loss in the network forwarding IP stack. We are currently modifying the fkt source code [3] with additional probes. With this we expect to find a number of bottlenecks and areas where packets are lost. Preliminary findings show that valid packets are most likely to be dropped initially at the *NIC*.

A solution to this, *hardware fault*, would be to upgrade the 10 Mb Ethernet cards to a 10/100 Mb Ethernet cards, as we are interested in the software failures not that of hardware. By extensively verifying the operation of the software from the *vortex* functions, handling of *incoming device driver* functions, *IP* functions, through to the *outgoing device driver* functions we will derive a formal description for validating and verifying the system.

The Invisible Recovery technique has proven a reliable way in which to provide fault tolerance to network software protocols and is currently being tested for possible use in a wider range of communications protocols including those in use in aerospace systems.

References

- [1] Stevens, W. R., UNIX Network Programming, Vol. 1, 2 Ed., Prentice Hall, 1997.
- [2] Mitchell, G.G., and Brown, S., An Approach for Network Communications Systems Recovery, Applied Informatics 2000, Innsbruck, Austria, pp 575-578, 2000.
- [3] Russell R.D., and Chavan, M., Fast Kernel Tracing: A Performance Evaluation Tool for Linux. Applied Informatics 2001, Innsbruck, Austria, 2001.
- [4] Kuenning, G.H., Precise Interactive Measurement of Operating Systems Kernels. Software - Practice & Experience 25(1): 1-21, 1995.
- [5] Tamches, A., and Miller, B.P., Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels, proceedings of the 3 Symp. on Operating Systems Design and Implementation, pp117-130, ACM, 1999.
- [6] Johnson, D.B., Distributed System Fault Tolerance Using Message Logging and Checkpointing. PhD thesis, Rice University, 1989.
- [7] Elnozahy, E.N., M., Alvisi, A., Wang, Y., Johnson D.B., A Survey of Rollback-Recovery Protocols in Message-Passing Systems, School of Computer Science Carnegie Mellon University, Pittsburgh, PA 15213 USA. CMU-CS-99-148, June 1999.
- [8] <http://www.snifferpro.co.uk/>
- [9] <http://www.kernel.org/pub/>
- [10] <http://www.linuxdoc.org/LDP/tlk/tlk-title.html>
- [11] Almesberger, W., Linux Network Traffic Control - Implementation Overview, EPFL ICA, April 23, 1999
- [12] Floyd, S., and Fall, K., Router Mechanisms to Support End-to-End Congestion Control, LBL Technical report, February 1997.
- [13] Mitchell, G.G., and Brown, S., A Model for Invisible Recovery Applied to Communications Networks, Applied Informatics 2001, Innsbruck, Austria, pp76 -81, 2001.
- [14] AIX Version 4.3 System Management Guide: Communications and Networks, IBM Corporation, USA.