

Distributed IPC using Virtual Device Driver in Monolithic Kernel

Susmit Bagchi

Department of Informatics, Gyeongsang National University, Jinju 660-701, South Korea
susmitbagchi@yahoo.co.uk

Abstract—The applications of distributed computing are ubiquitous ranging from client/server based Internet systems to the cluster or grid and cloud computing systems. The distributed applications require efficient and transparent distributed interprocess communication (IPC) mechanism as a service from the operating systems supporting those applications. This paper describes the designed architecture and implementation of kernel-level distributed IPC mechanism using device driver framework offering transparency and high performance. The implementation is made in Linux monolithic kernel and the performance is measured by conducting the experiments in real-life execution environments. The experimental data illustrate that the distributed IPC in Ethernet network takes 629 μ s for small messages (25Bytes) and 2929 μ s for relatively large messages (5KB). The experiments in wireless network achieve 34528 μ s and 42511 μ s for the small and large messages, respectively. On the average, the designed architecture achieves distributed IPC time as 0.899 μ s/Byte in Ethernet and 92.628 μ s/Byte in wireless network environments. The proposed architecture and implementation offer 12.73% ~ 75.84% enhanced performance as compared to other distributed IPC models depending upon data length.

Keywords - IPC; RPC; monolithic kernel; microkernel; device driver.

I. INTRODUCTION

The large scale distributed computing applications require the efficient and transparent distributed interprocess communication (IPC) as a service from the underlying operating systems. Apart from the distributed applications, the distributed operating systems also require the mechanism to achieve distributed IPC having high-performance [1]. In general, the existing distributed IPC models are designed as a middleware. However, the middleware architecture fails to deliver high-performance due to instruction dilation. The kernel-level architectures and models are generally favoured in view of reducing overhead and application development complexity as well as to achieve high performance [1, 2]. The kernel-level distributed IPC mechanisms are employed to realize distributed synchronization, remote tasking and distributed shared memory [1, 2, 3, 4]. Due to the availability of 3G/4G wireless networking with high bandwidth and reliability, a subset of the distributed computing applications are getting transformed into mobile computing applications.

The mobile computing applications need the user-transparent, reliable and efficient distributed IPC in wireless network environment. In a novel approach, this paper proposes the use of device driver framework of kernel to realize the kernel-level distributed IPC system offering location transparency and reliability. This paper describes the architectural design and implementation of kernel-level distributed IPC mechanism suitable to wired as well as wirelessly networked systems. The monolithic kernel offers enhanced performance of the overall system including the kernel subsystems [1, 2, 5]. The architecture, proposed in this paper, considers the monolithic kernel address space in order to achieve two goals such as, (1) *enhancement of performance of distributed IPC* and, (2) *adapting the operating system kernel to distributed computing as well as mobile computing environments*. The architecture, proposed in this paper, considers the device driver framework of monolithic kernel address space of Linux operating system. The implementation is made as a set of kernel module and associated daemon processes.

The performance of the system is evaluated through experiments conducted in real-life execution environments. The experimental data illustrate that the proposed architecture achieves 0.899 μ s/Byte and 92.628 μ s/Byte of distributed IPC time in Ethernet and wireless network environments, respectively. The proposed design and implementation offer 12.73% to 75.84% improvements of the performance of distributed IPC mechanism as compared to other existing models in case of Ethernet depending upon data length. In case of wireless network environment with varying available bandwidth as 20% ~ 47% of 100Mbps peak (with 25% signal strength), the proposed architecture and implementation achieve distributed IPC latency to be 34528 μ s for short messages (25Byte) and 42511 μ s for large messages (5KB). The architecture is easily realizable within the Linux kernel and offers user-transparent, reliable and efficient distributed IPC service in wire and wireless networked distributed computing systems.

The rest of the paper is organized as follows. Section II describes the design of architecture and internals. Sections III and IV explain implementation and experimental evaluation, respectively. Section V describes related work and section VI depicts a detailed comparative analysis. Section VII concludes the paper.

II. THE ARCHITECTURAL DESIGN

The distributed IPC mechanism can be designed either in user address space as a middleware or as a subsystem in the kernel address space. The middleware offers easy portability at the cost of overall performance of the system. On the contrary, the kernel-level subsystems offer improved performance and system security. The kernel-level distributed IPC architecture can be viewed as a software layer spanning over a set of computing nodes connected by network as illustrated in Figure 1.

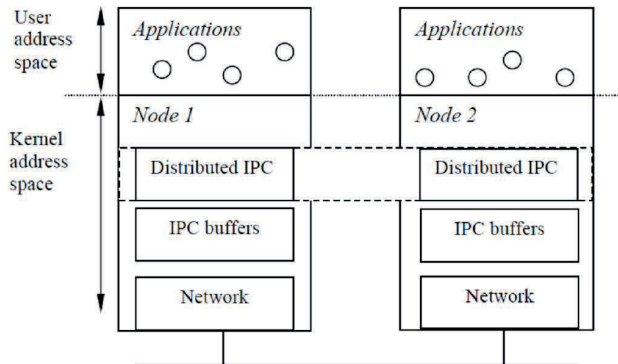


Figure 1. The Model of Kernel-level Distributed IPC.

At each node, the message buffers are maintained by the individual kernel for holding IPC data. The kernel-level distributed IPC architecture creates a system-wide virtual layer of service to user applications offering three advantages such as, (1) *location transparency*, (2) *uniform interface* and, (3) *efficiency*. The location transparency is offered to the user space applications by the distributed kernel modules by constructing the system wide view of processes executing at different nodes, where a kernel-level process name at a node is a tuple defined as $\langle \text{Node_ID}, \text{IP}, \text{Process_name} \rangle$. The user space applications invoke the IPC through the uniform IO interface offered by the device driver framework of monolithic kernel. The distributed kernel modules at different nodes communicate through the inter-module communication channels over the network. The internals of architectural design of a kernel-level distributed IPC module is illustrated in Figure 2. The distributed IPC subsystem of a node can be decomposed into two components namely, the *IPC module* and the *daemons*. The daemons are processes responsible to maintain inter-module distributed communications using the channels. A distributed IPC module within the kernel space offers a uniform IO interface to the distributed computing applications residing in the user address space by using *open/read/write/close* interface of the device drivers. The distributed IPC modules maintain two interaction pathways in kernel such as, an *IO path* and a *control path*. The IO path

encompasses the IPC module, daemons and applications implementing packet-based IO between subsystems. However, the control path encompasses the IPC module and daemon processes implementing internal IO control mechanisms.

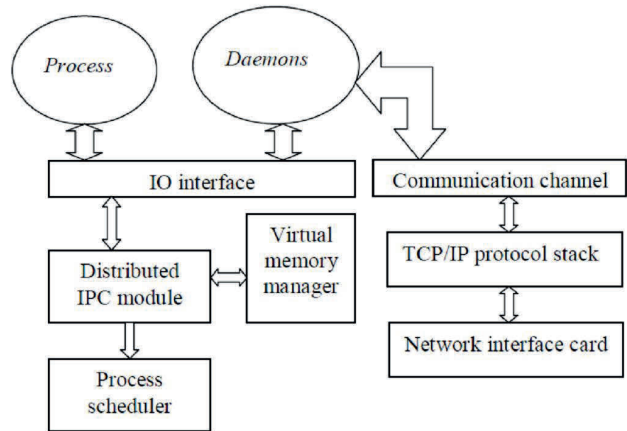


Figure 2. The Internals of Designed Architecture.

The distributed IPC messages are designed as packets having fixed size containing IPC data of varying length. The IPC packet format is illustrated in Figure 3.



Figure 3. The Distributed IPC Packet Format.

The header of the distributed IPC packet has fixed length and contains IO command and flag indicating preferences related to synchronization of communication. The IPC data within the packet may have varying length from short IPC data (a few Bytes) to large IPC data (in the order of KB).

As a design rationale to offer greater flexibility to the user applications, the IPC message packets are exposed to the distributed computing applications so that the applications can construct and interpret the packet contents. The distributed applications residing in user space need not to enquire and locate the distributed processes in a distributed system in order to deliver the IPC messages. The kernel-level IPC modules at the distributed nodes and the daemons maintain the system-wide view of distributed processes and their locations along with communicating addresses. Hence, the user applications invoke an IPC request with location transparency. If the destination process is indeed a local process, then the message is delivered locally by the IPC module of the kernel. Otherwise, the message will be delivered to the remote process through channels by the daemons and the remote IPC modules.

III. IMPLEMENTATION

The architecture is implemented as a combination of kernel module and a set of daemons in Linux 2.6 kernel having Fedora core. The kernel module is designed and implemented based on the architectural framework of character device driver. In this case, the kernel module can be considered as a virtual device driver (VxD) because, the kernel module does not control any physical device. The daemons are the processes dynamically created by the VxD during the startup and module initialization by the kernel. The VxD communicates with the user applications and the daemon processes through the command IO interface (CII). The internals of CII and associated command-packet processing are illustrated in Figure 4.

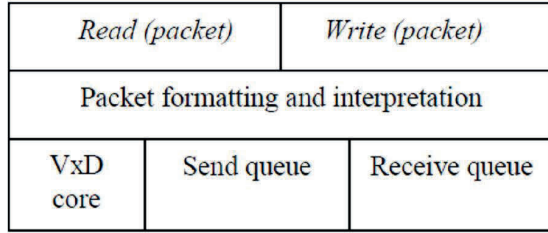


Figure 4. The Command IO Interface and Internal Organization.

The communication is carried out using a set of packets containing commands and corresponding reply. A command is sent by the user application or daemon process to the VxD through CII. In turn, the VxD sends the reply back to the processes in the reply packet. The write interface of VxD is used by the applications or daemons to send command packets to the VxD and the read interface of VxD is used by the applications or daemons to receive the reply as well as data from the VxD. The packet structure used in CII is given in Figure 5.

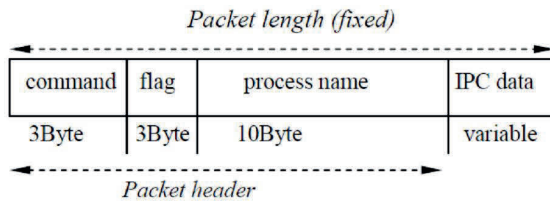


Figure 5. The IPC Packet Structure.

The packet length equal to MTU (Maximum Transmission Unit) is used to reduce the packet fragmentation and reassembly in order to improve performance (however experiments are carried out with varying lengths for performance evaluations). Thus, the size of the packet is kept fixed having sufficiently long data-space to store IPC data of

varying length. The command packets contain a fixed length header and a fixed length space for destination process address. A destination process name is indicated as a string (Process_name) by a sender process in the user space. However, the corresponding kernel-level process name is constructed as a tuple given by, $\langle \text{Node_ID}, \text{IP}, \text{Process_name} \rangle$ determining as well as locating a distributed process uniquely in a system. The remaining of a packet contains the IPC message data of variable length. The header of the packet contains various commands and the associated flags indicating whether the IPC is synchronous or asynchronous. The descriptions of various commands are illustrated in Table 1.

TABLE 1. SET OF COMMANDS AND THE DESCRIPTIONS.

Command	Descriptions
<i>Register_ProcName</i>	To register process name to VxD
<i>Send_IPC</i>	To send IPC packets to another process
<i>Recv_IPC</i>	To read a IPC packet from another process
<i>Send_Event</i>	To send an event to another process
<i>Recv_Event</i>	To receive an event from another process
<i>Deregister_ProcName</i>	To deregister from VxD
<i>Send_BRDCST</i>	To broadcast a message or event

The registration command is used by the application processes to register to VxD in order to use distributed IPC service of the kernel. The registration system enables the VxD to register the process name and to allocate virtual memory area (VMA) to send and receive IPC messages from local or remote processes. A process can deregister from VxD and in such case, VxD removes the entry of respective process name from the process-list and de-allocates the kernel memory pages held by the process as buffers. The daemons are created by VxD and are not required to register to VxD. The communication between two processes can be either IPC messages or events. The events are distinguished from IPC messages, where the events are indicated by a set of predefined flags. The processes are identified by VxD using the related file descriptor entry (f_dentry) of kernel data structure of a device file. The internal data structure maintained by a VxD is illustrated in Figure 6.

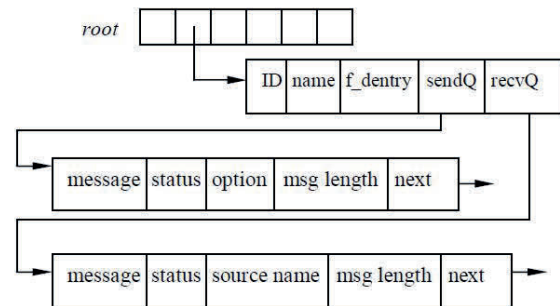


Figure 6. Data Structure used by Distributed IPC module.

In order to reduce memory underutilization due to page-locking, the VxD allocates resources from kernel atomically depending upon IPC requests. The daemon processes are the communicators, which communicate to other remote daemons through communication channels over TCP/IP protocol stack. The daemon processes perform two activities such as, (1) *creating a communication channel between two distributed VxDs residing at two nodes connected by network* and, (2) *performing data IO between two nodes over the network*. The implementation of VxD and daemons are carried out in C language and the distribution of lines of codes (LOC) is given in Table 2.

TABLE 2. DISTRIBUTION OF LOC IN SUBSYSTEMS.

Software subsystem	Lines of code (LOC)
VxD	145
Daemon (sender)	118
Daemon (receiver)	112

IV. EXPERIMENTAL EVALUATIONS

The proposed architecture is implemented in Linux kernel 2.6 having Fedora core and the experiments are carried out in real-life execution environments in two steps namely, Experiment-I and Experiment-II. The configurations of systems are given in Table 3, where ISP stands for Internet Service Providers indicating different network domains hosting the nodes contain distributed processes. Experiments are carried out in ten iterative rounds and the average values are measured using bidirectional data IO with varying length from short messages (packet size $\leq$ MTU) to relatively large messages crossing the 4KB page boundary of the kernel (packet size $>$ MTU). The time is measured using the CPU clock at the nodes having microsecond resolution.

TABLE 3. SPECIFICATIONS OF THE SYSTEMS.

Nodes/network	Specification
Node 1	CPU: Intel P9300; Clock: 2.26GHz RAM: 2GB; HDD: 128GB SSD; Network: 100Mbps Ethernet, 100Mbps WLAN
Node 2	CPU: Intel T9550; Clock: 2.66GHz RAM: 2.96GB; HDD: 256GB SATA; Network: 100Mbps Ethernet, 100Mbps WLAN
Network	Ethernet: 100Mbps LAN (ISP 1 and ISP 2); Wireless: 100Mbps WAP; Node2-WAP 200m; Signal strength: 25%

A. Experiment-I

Experiment-I is carried out in Ethernet networking environment. The experimental setup is illustrated in Figure 7. The bandwidth of the Ethernet network is 100Mbps at peak and 75 ~ 80 Mbps on average. The node 1 and node 2

are connected through two different Internet service providers (ISP) residing in two different domains and the geographic distance between the nodes is nearly 1km.

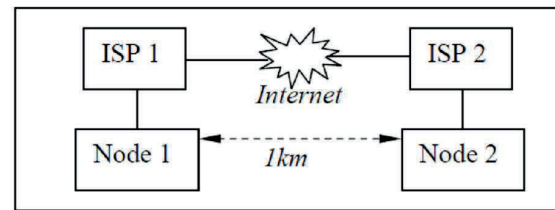


Figure 7. Setup of Experiment-I, ISP: Internet Service Provider.

A process at node 1 sends a series of IPC messages to another process at node 2 and vice-versa, where the length of IPC data varies from 25Byte to 5KB. The distributed IPC latency values are measured in μ s resolution. The experimental results are illustrated in Figure 8.

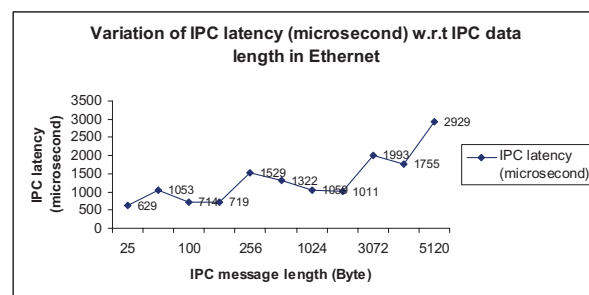


Figure 8. Variation of IPC Latency and IPC Message Length in Ethernet.

It is evident from Figure 8 that, the peak-to-peak IPC latency is increased by 2300 μ s for the corresponding increase of message length of 5115Byte. Thus, the min-to-max rate of increment of distributed IPC latency is 0.449 μ s/Byte. In addition, the average value of variation of distributed IPC latency with respect to data length in Ethernet network is 0.899 μ s/Byte.

B. Experiment-II

The Experiment-II is carried out in wireless Internet environment as illustrated in Figure 9. In this experiment, the node 2 is connected to wireless Internet through a WAP (wireless access point) and two distributed processes (residing at node 1 and node 2) exchange a series of IPC messages of varying lengths. The geographic distance between WAP and node 2 is measured to be 200m. The wireless Internet is made publicly accessible through WAP in order to load the wireless network with random burst of data traffic.

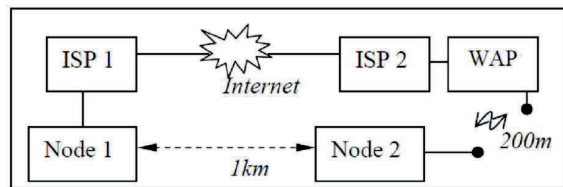


Figure 9. Setup of Experiment-II, WAP: Wireless Access Point.

According to observation during experiments, the available wireless bandwidth is randomly varied as 20% ~ 47% of peak (100Mbps) with signal strength equal to 25% and the WAP is publicly shared by multiple users. The experimental results giving measured distributed IPC latency values are illustrated in Figure 10.

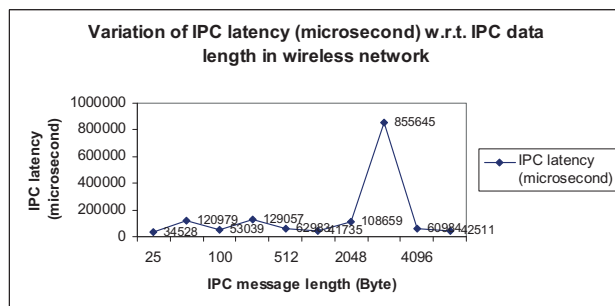


Figure 10. Variation of IPC Latency and IPC Message Length in Wireless Network.

It is evident from Figure 10 that, variation of distributed IPC latency with respect to message length is random. However, the average value of distributed IPC latency is 92.628 μ s/Byte. The min-to-max variation of IPC latency is 23.12% due to monotonic increment of IPC data length from 25Byte to 5KB, which enforces the crossing of page-boundary in VMA of the kernel. Hence, the distributed IPC latencies offered by the proposed architecture and implementation in Ethernet and wireless networks environments are not sufficient to significantly degrade the overall performance of distributed computing.

V. RELATED WORK

The IPC mechanism is a fundamental building block of operating systems. The basic concepts and models of IPC mechanism are extended to the distributed computing systems [1, 2]. There exist two approaches to achieve the distributed IPC namely, middleware-based approach and kernel-based approach. The system V IPC mechanism is extended to realize the distributed IPC in the Linux operating system [1]. In this model, the distributed synchronization mechanism is realized at the kernel level to reduce overhead.

However, the designed architecture contains the single point of failure due to a central-server based model construction. Researchers have proposed the monitor based distributed IPC mechanism [6] and the support of operating system to implement distributed communication in cluster environment [7]. A majority of the distributed IPC designs consider middleware-based architecture either in part or in full. The main disadvantage of middleware-based distributed IPC models is reduced performance due to instruction dilation following enhanced call-depth. However, the kernel-level design approaches benefit from faster response, high performance and greater security [1, 2, 5].

The KDIPC is a kernel-level IPC library to support distributed shared memory and distributed semaphores [2]. However, it is argued that concurrency control and distributed synchronization based on shared networked files are complex [2, 8] and a protocol is designed to achieve sequential consistency in NFS [9]. The distributed IPC mechanism is also used to implement remote tasking [3, 10]. The distributed IPC based remote tasking is realized using Mach microkernel in two different versions namely, NORMA and DIPC [3, 11, 12]. The performances of NORMA and DIPC are comparatively lower than the other monolithic kernel based designs. One reason is that the microkernel architecture incurs heavy context switching and thereby enhancing latency [5]. The Peregrine is a high-performance remote procedure call (RPC) design based on the fast distributed IPC system [4]. The latency of distributed IPC during null-RPC in Peregrine is lower than the full-RPC involving larger packet sizes due to the interpretation of data stored in the packets and packet fragmentations.

VI. COMPARATIVE ANALYSIS

The distributed IPC system is widely employed to realize the RPC mechanism. The Peregrine RPC system is a high-performing mechanism, which implements a very light-weight protocol for delivery of IPC messages [4]. In another approach, the distributed IPC is implemented in Mach microkernel in order to achieve extensibility and transparency [3]. However, the architecture proposed in this paper (Dist. IPC) employs monolithic kernel architecture. The comparison of distributed IPC latency values for null-RPC (RPC contains no argument and return values) and null-IPC (zero byte IPC data) is illustrated in Figure 11.

It is evident that, performance of Peregrine model is comparable to the proposed architecture (i.e. Dist. IPC) of this paper, whereas NORMA-Paragon and DIPC (Paragon and SCSI) models dilate the distributed IPC latency values significantly. In addition, the proposed Dist. IPC architecture offers the lowest latency value equal to 500 μ s.

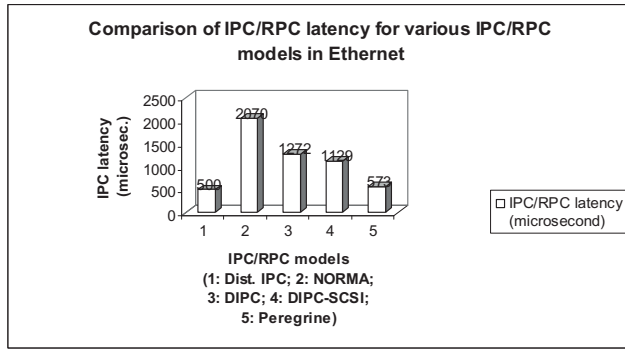


Figure 11. Comparative Analysis of IPC/RPC Latency for Different Models.

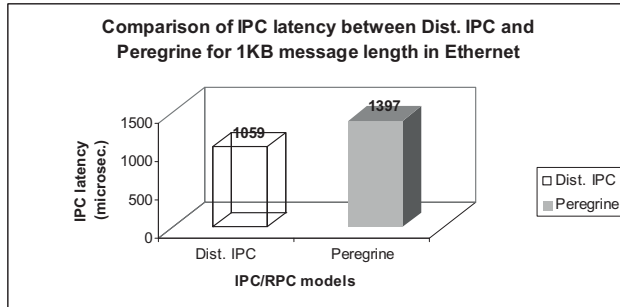


Figure 12. Comparison of IPC Latency between Dist. IPC and Peregrine in Ethernet.

The comparison between distributed IPC latency values of Peregrine and the proposed architecture is illustrated in Figure 12, where the packet length is 1KB. According to Figure 12, the proposed architecture performs significantly higher than the Peregrine model in case of moderately large sized packets.

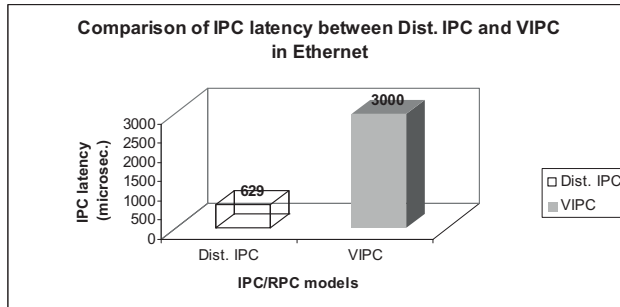


Figure 13. Comparison of IPC Latency between Dist. IPC and VIPC in Ethernet.

In another approach, the extended system V IPC model (VIPC) of distributed interprocess communication is proposed based on monolithic kernel of Linux [1]. The

comparison of performance of VIPC and the proposed Dist. IPC architecture is illustrated in Figure 13. The distributed IPC latency of VIPC is significantly higher than the performance of designed architecture proposed in this paper. In addition, the VIPC design model contains the single-point of failure, because a central-server based design approach is employed for name resolution and message routing. However, the architecture proposed in this paper does not employ any central-server based message routing and name resolution methods.

VII. CONCLUSION

The efficient design of distributed IPC is a key to realize low latency remote tasking, distributed shared memory, distributed synchronization and message-based mobile computing. The monolithic kernel based design of distributed IPC architecture can offer high performance and low latency. The efficient distributed IPC systems can be used to implement event-based distributed computing systems. This paper proposes the architecture and implementation of kernel-level distributed IPC mechanism. The design considers monolithic Linux kernel and the model of virtual device driver. The implementation of the virtual device driver realizing distributed IPC follows the framework of character device of monolithic kernel. The experiments are carried out in real-life execution environments in Ethernet and wireless Internet connecting geographically distributed nodes. The experimental results illustrate that the proposed architecture achieves distributed IPC latency $629\mu s \sim 2929\mu s$ in Ethernet and $34528\mu s \sim 42511\mu s$ in wireless networks depending upon message length. However, the proposed architecture offers from 12.73% to 75.84% improvement in performance as compared to other distributed IPC/RPC systems in Ethernet.

REFERENCES

- [1] Peng X., Yuanyuan L., Wu D., A Model of Distributed Interprocess Communication System, 2nd International Workshop on Knowledge Discovery and Data Mining, IEEE, 2009.
- [2] Tapus C., Hickey J., Distributed Synchronization with Shared Semaphore Sets, 2005 IEEE International Symposium on Cluster Computing and the Grid, IEEE, 2005.
- [3] Milojicic D. S., Langerman A., Black D. L., Sears S. J., Dominijanni M., Dean R. W., Concurrency, a Case Study in Remote Tasking and Distributed IPC, 29th Annual Hawaii International Conference on System Sciences, 1996.

- [4] Johnson D., Zwaenepoel W., The Peregrine High-Performance RPC System, *Software-Practice and Experience Journal*, Vol. 23(2), 1993.
- [5] Tapus C., Smith J. D., Hickey J., Kernel level Speculative DSM, *IEEE International Symposium on Cluster Computing and the Grid (CCGRID'03)*, IEEE, 2003.
- [6] Ammann E. M., DIPC – A Monitor for Distributed Interprocess Communication, *3rd Euromicro Workshop on Parallel and Distributed Processing*, pp. 272, 1998.
- [7] Diaz A. F., Ortega J., Fernandez F. J., Anguita M., Canas A., Prieto A., An Efficient OS Support for Communication on Linux Clusters, *2001 International Conference on Parallel Processing*, IEEE, 2001.
- [8] Pawlowski B. et al., The NFS Version 4 Protocol, *2nd International System Administration and Networking Conference (SANE 2000)*, 2000.
- [9] Tapus C., Nogin A., Hickey J., White J., A Simple Serializability Mechanism for a Distributed Objects System, *17th International Conference on Parallel and Distributed Computing Systems (PDCS 2004)*, ISCA, 2004.
- [10] Milojicic D., Black D., Sears S., Operating System support for Concurrent Remote Task Creation, *9th International Parallel Processing Symposium*, USA, pp 273-290, 1995.
- [11] Bryant B., Langerman A., Sears S., Black D., NORMA IPC: A Task-to-Task Communication System for Multicomputer Systems, *OSF RI Operating Systems, Collected Papers*, Vol. 2, 1993.
- [12] Langerman A. et al., NORMA IPC Version Two: Architecture and Design, *OSF RI Operating Systems, Collected Papers*, Vol. 3, 1994.