PAPER Special Section on Internet Technology IV

Linux IPv6 Stack Implementation Based on Serialized Data State Processing

Hideaki YOSHIFUJI[†a)], ***Student Member***, **Kazunori MIYAZAWA**[††b)], **Masahide NAKAMURA**[†††c)], **Yuji SEKIYA**[†d)], ***Nonmembers***, **Hiroshi ESAKI**[†e)], ***and*** **Jun MURAI**[††††f)], ***Members***

SUMMARY IPv6 is realized as the next generation internet platform, succeeding the current IPv4 internet environment. Linux, one of the major operating systems, has supported IPv6 since 1996, however, the quality of the protocol stack has not been good enough for professional operation. In this paper, we show our IPv6 stack implementation design regarding the neighbor management in Neighbor Discovery Protocol (NDP), the routing table management and the packet processing using XFRM architecture. The implementation is designed based on the Serialized Data State Processing, which aims at simpler object management so as to achieve stable, flexible and extensible IPv6 stack. According to the TAHI IPv6 Protocol Conformance Test Suite, we can show our implementation achieves enough implementation quality.

key words: *IPv6, Linux, serialized data state processing*

1. Introduction

The Internet has run with the Internet Protocol Version 4, so called as IPv4 [11], since the end of 1960s. At the end of 1980s, the internet experts working at the IETF (Internet Engineering Task Force) [18] has recognized that we need a new version of Internet Protocol to come up with too rapid growth of the Internet. In 1992, this new version of protocol was named as IPng (IP next generation). The primary concerns since the middle of 1990s were the significant growth of routing table entries maintained in the routers and the throughput yielded by the routers. Furthermore, there was some concerning on the shortage of IP (IPv4) addresses to be allocated to.

The full-scale technical discussion on the IPng started in 1992 at the IETF. IPng, i.e., Internet Protocol Version 6 (IPv6) [2], was designed to solve the various issues on the traditional IPv4, such as performance of packet forwarding, protocol extensibility, security and privacy. The basic specification of IPv6 was defined in 1994. After the experimental implementation and network operation (e.g., 6bone [1]), the IPv6 technology is now getting into the production and professional phase. Commercial IPv6 services by Internet Service Providers and the applications running with IPv6 has been already available around us. This means that, the IPv6 stack implemented in any devices must be of production quality.

Linux system has also supported the IPv6 protocol as well as other operating systems such as FreeBSD [17], Sun Solaris [15] and Microsoft Windows XP [9]. Linux has included IPv6 stack since 1996 when early Linux 2.1.x version released. After the integration of IPv6 stack into the mainline kernel, however, it has not been actively developed nor maintained by the kernel maintainers.

Through the observation and analysis of the legacy IPv6 stack implementation in Linux, we realized that the implementation architecture and design of the legacy IPv6 stack were rather complex and were not well organized. In this paper, we propose a new implementation design based on the Serialized Data State Processing approach. The proposed design is simple and extendable, by means of the introduction of serialized object and state processing structure. The implementation described in this paper has been integrated as the USAGI Project IPv6 stack [20].

We describe our IPv6 stack implementation design and the conformance evaluation result in this paper. Section 2 discusses the background, related works and abstract of our design principle. Section 3 describes the neighbor management architecture in NDP (Neighbor Discovery Protocol) implementation, Sect. 4 describes the routing table management in router, and Sect. 5 describes IP packet processing using XFRM concept. Finally, Sect. 6 gives a brief conclusion.

2. Serialized Data State Processing

2.1 Background and Problem Description

It is well-known that the state management of large system is not an easy task. In general, we usually divide the system into several parts, so called "modules," and prohibit the direct operations against values in the module from the other modules. One of the methods to achieve the goal using distributed self-managed objects and message passing among them is Object Oriented Programming (OOP). It is also important for efficient multi-processing to divide system into

Manuscript received June 25, 2003.
Manuscript revised September 19, 2003.

[†]The authors are with the University of Tokyo, Tokyo, 113-8656 Japan.
[††]The author is with Yokogawa Electric Corporation, Musashino-shi, 180-8750 Japan.
[†††]The author is with Hitachi Communication Technologies, Ltd., Yokohama-shi, 244-8567 Japan.
[††††]The author is with Keio University, Fujisawa-shi, 252-8520 Japan.
a) E-mail: hideaki@yoshifuji.org
b) E-mail: Kazunori.Miyazawa@jp.yokogawa.com
c) E-mail: masahide_nakamura@hitachi-com.co.jp
d) E-mail: sekiya@wide.ad.jp
e) E-mail: hiroshi@wide.ad.jp
f) E-mail: jun@wide.ad.jp

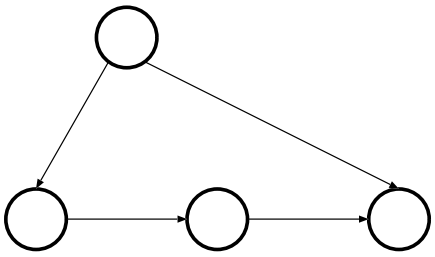


Fig. 1 Data object dependency graph.

multiple modules, which autonomously run from states of other modules.

Linux system has obviously been applied the modularization, i.e., a kind of OOP. For example, the implementation of reference counting, which is defined to manage the lifetime of each object, is based on the idea of OOP. However, the state dependency among the objects would not be well organized. It is required to have complex mutual exclusion and rather exception handing due to the collision of dependency and state transition. It is also difficult to extend features and functions. This is because the complex state dependency among objects makes difficult the correct state management and implementation, against the introduction of new object(s) or state(s) into the existing system.

In this paper, we reorganize the state transition of objects in the Linux IPv6 stack. In Linux IPv6 stack, there are many state transition topologies, which has a loop in the state transition, as shown in Fig. 1. When we do not have any loop in the state transition topology, the state management and implementation becomes far easier than the case where there is a loop in the state transition. We name this design principle as "Serialized Data State Processing." With this approach, we can achieve stable, flexible and extensible implementation.

2.2 Serialized Data State Processing

Large processing task is divided into multiple small processing tasks and objects. These tasks and objects are connected as linear as possible, in order to avoid including a loop in the state transition topology. The basic entity of the Serialized Data State Processing design is self-managed data object and consists of the following elements:

Mutual Exclusion
The serialization of accesses to the object is implemented using lock or semaphore. In general, from the view point of serialization of data access, the mutual exclusion is unnecessary inside the object if the container has a mutual exclusion mechanism. However, we introduce it inside the object in order to make objects as independent as possible each other. With this implementation design, it is expected that we can share each object between multiple containers, and that we can improve the performance efficiency in multiprocessing environment.

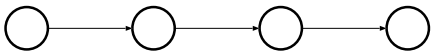


Fig. 2 Serialized data state.

Reference Counter
Reference counter manages the lifetime of the object based on the number of object references. An object can kill itself if its reference count becomes zero. However, with some complex state management, we need an external garbage collection mechanism to remove non-referenced objects in the container. When we separate the management of reference counter for each object and the management of objects in the corresponding container, we can manage the objects more easily and surely than we can without this separation.

In addition to these elements, we need an object management timer, which is included inside the object if applicable.

Serialized Data State Processing is a kind of combination of self-managed data object described above where one or more data objects are combined by directed links, and then the set of objects work all together. When we define the topology and object appropriately, we can eliminate the unnecessary (external) referencing against the other object while maintaining the object independency. In particular, when we can define the linear topology (as shown in Fig. 2), we can eliminate the conflicts of resource dependency which frequently cause a deal lock.

3. Neighbor Discovery

Neighbor Discovery (ND) is one of fundamental elements of IPv6 protocol suite. ND has the following functions [10], [19].

- Router and Prefix Discovery
- Address Resolution and Neighbor Unreachability Detection
- Redirect

Address Resolution and Neighbor Unreachability Detection maintain the status of neighbors and this is the core functionality of ND, i.e. router selection should be executed in conjunction with the status of neighbor nodes. The state of each neighbor is maintained via the Neighbor Cache Entry (NCE). Therefore, in order to maintain the stable communication, it is required to have the accurate timer and state management of NCE against the various events expected to occur on the network.

In the legacy Linux implementation, NCE was managed by the global periodic polling timer (Global Timer) and by the timer maintained inside the entry (Internal Timer), as shown in Fig. 3.

Global Timer
Periodic timer (30 seconds) invokes the management task, which checks reachability of every node and cleans up entries in the table.

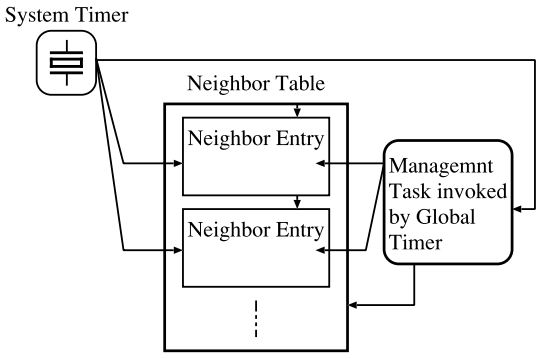


Fig. 3 Legacy Linux NDP table management.

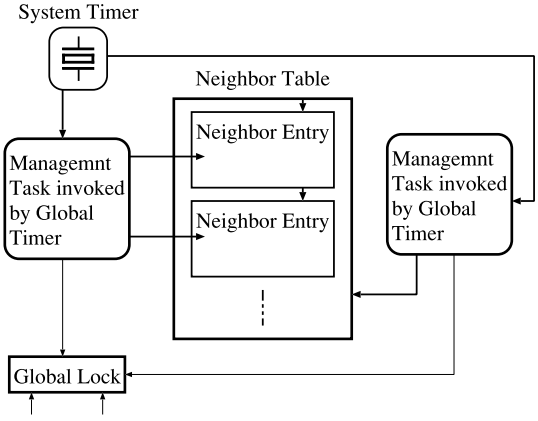


Fig. 4 KAME NDP table with periodic timer.

Internal Timer

Dynamic timer residenting in the entry invokes the management task for itself in semi-reachable states.

The state of each NCE is referred both by the Global Timer and by the Internal Timer. This means that status management operation is complex and conflicted. At the same time, though the protocol requires the accuracy of few seconds for all state management, the Global Timer can not provide sufficient time accuracy. This is because Global Timer runs with a order of decades seconds (30 sec.) accuracy.

NDP table management in KAME uses the periodic timers. One is for status management and the other is for table management. The timers run management tasks in every one second respectively (Fig. 4) so that the error of timing accuracy becomes 1 second at worst. However, this table management design in KAME stack would not be optimal implementation, especially for multi-processing environment. This is because it uses the "global" locking mechanism. It is worse when frequent timing refinement is required.

Based on the above observations, we redesigned the NCE management implementation based on our proposed design. We reorganized the management tasks including timers and mutual exclusion (locks) for state management.

As shown in Fig. 5, we separate the NCE management, that is invoked only by dynamic Internal Timer. Also, in the

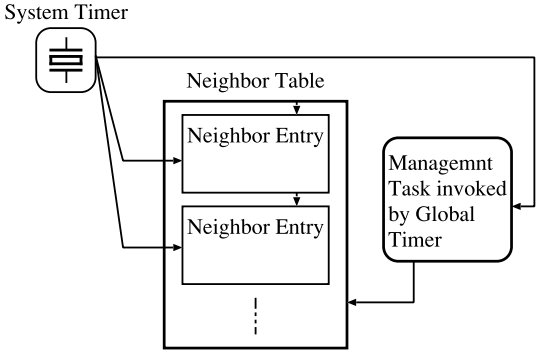


Fig. 5 Proposed NDP table with dynamic timers.

Table 1 TAHI conformance test result of NCE management (PASS ratio).

Linux 2.4.18	USAGI 2.4	KAME/FreeBSD 4
39%	89%	98%

NCE management, each Internal Timer residenting at each entry is responsible only to an corresponding entry. This means that the NCE management interacting with Internal Timer is autonomous and self-managed operation at each entry. The management task, that is the garbage collection, invoked by the Global Timer, never directly refer any NCE in the table.

In summary, with the proposed implementation design, the resource management, including mutual exclusion, and the control frequency of global clocks can be simplified.

Table 1 shows the conformance evaluation results via the TAHI [16] IPv6 Conformance Test Suite. As shown, our NDP implementation achieves higher conformance compared with the legacy Linux implementation.

4. State Processing in Routing Table Management

Routing is the process of delivering IP packets to the destination node. Routing table maintains all the information necessary to forward the received IP packet either to next hop router or to the destination node.

Linux IPv6 routing table, known as "Forwarding Information Base," is constructed by the Radix Tree [14], which is composed by nodes and leaves.

Figure 6 shows the routing table and neighbor management in Linux implementation. Every node represents a bit position to examine. Each node with the `RTN_RTINFO` flag has a linked list for the leaves, which represents the actual routing information, such as metric and next-hop. The next-hop then actually refers to the corresponding NCE. Unlikely to the traditional BSD variants, Linux supports multiple paths for a single destination. Leaves are chained in numerical order of metric.

In this section, we describe how the Serialized Data State Processing approach is applied to the routing management in Linux system. The routing management includes (1) default routers management and (2) router precedence management. As for the list of default routers, we eliminate

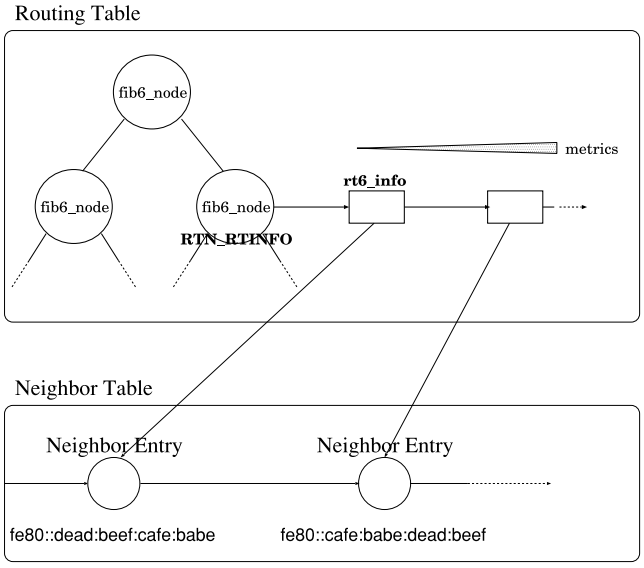


Fig. 6 Linux routing table and neighbor table.

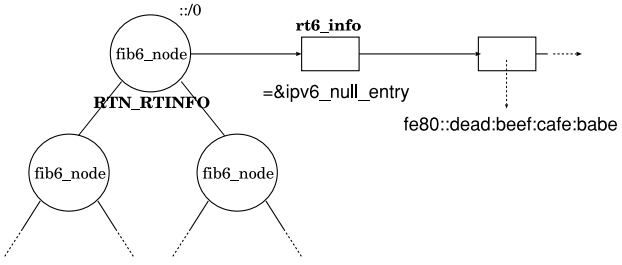


Fig. 7 Linux IPv6 routing table structure.

the "Default Router List" by means of putting the information with regard to default routers as the routing information for ::/0. As for the Default Router Selection and Load Sharing between Routers, we use the link topology to describe the state for round-robin. By using this mechanism, we can eliminate the link from external entity for maintaining priority information. In addition, we can achieve load sharing between routers associated with generic routes.

4.1 Default Routers Management

ND introduces a new data structure named as "Default Router List." It contains the information of "default routers" advertised via Router Advertisement messages generated by neighbor routers.

KAME maintains the information of default routers in separate "Default Router List." On the other hand, with legacy Linux implementation, the Default Router List is held on the top-level root of the routing table tree as if it is a kind of routes to ::/0 ("default route"). The first entry for ::/0 always points `ipv6_null_entry`, which is never used as normal routes, and the "default routers" follows it (Fig. 7). They are exceptionally used on hosts because of the special default router selection. When a default route is to be added, however, the corresponding information is

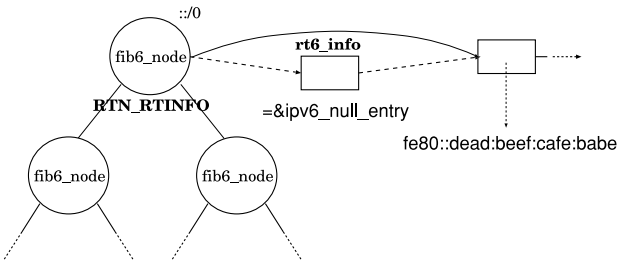


Fig. 8 USAGI IPv6 routing table structure.

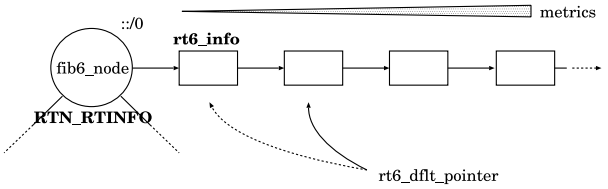


Fig. 9 Default routers in Linux.

attached at the next of the `rt6_info{}` structure which contains `ipv6_null_entry`. With this procedure, the appended routes to ::/0 can not be referred to, because the special default router selection is not applied to normal routes to ::/0.

When we investigate the principle of routing table and the concept of the "Default Router List," we realize that we should not treat ::/0 as a special case. In our proposed implementation, when we add a new routing entry on the top-level root of the tree, we replace the `ipv6_null_entry` with the new entry as shown in Fig. 8. When the last route is being deleted from the the top-level root of the tree, we re-insert `ipv6_null_entry`. It means that we actually treat the "default routes" which include the information regarding "Default Routers" as "normal" routes. As a result, we can naturally insert or remove the "default route" entries properly to or from the routing table, in the same way as we do for other normal routes.

4.2 Router Precedence Management

Picking up one router from the (next-hop) routers list, which has the same destination, is called "Router Selection." "Default Router Selection" can be considered as a special case for default routes.

As mentioned in the previous subsection, the default routers are stored on the top-level root node in the routing tree. In Default Router Selection, it applies a round-robin policy to pick up the available default router, when the currently selected default router becomes unreachable. To achieve this procedure in legacy Linux system, the default router was pointed by `rt6_dflt_pointer`, which is guarded by the global lock named `rt6_dflt_lock`, and updated when the current default router becomes unreachable (Fig. 9).

In this implementation, there were several problems described below.

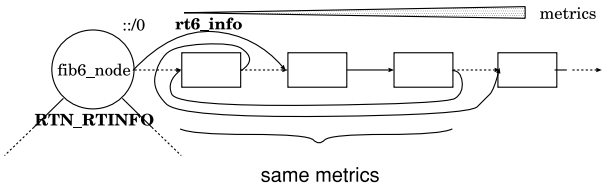


Fig. 10 New method for route round-robin.

Unfairness
`rt6_dflt_pointer` is reset when routing is modified; this happens very often and routers are not equally selected.

Lack of Generic Route Selection Mechanism
It is important to select appropriate route from multiple routes with some selection strategy and/or policy. The mechanism in "Default Router Preferences, More-Specific Routes, and Load Sharing" [3] would be a good candidate for it. Because `rt6_dflt_pointer` is static, single variable, it is only for the default routes (::/0) and the logic can not be applied to the generic route selections. We require a generic and appropriate mechanism to achieve it.

Metrics
Legacy Linux treats metrics for default routes, in other way than in normal routes. However, we rather propose to treat the default routes as normal routes. It is better to eliminate the special handling of metrics against the default routes.

To achieve this design goal, we introduced a new generic round-robin mechanism. We apply the round-robin for routes with the same metric, when a route in that set is used (Fig. 10). For the 2-bit "preference" of routes, which is advertised by routers, by means of the "Router Selection" proposal [3], we introduced new flags for the preference rather than we take up its reside in the metric, so that we can simplify other logic, e.g. routing table updating against the reception of RA message.

The proposed mechanism is as follows: When a route is looked up, we select the first entry with the highest preference in "probably reachable" state among the NCEs. Then, we move the selected route to the tail of the set of routes which are with the same metric as the selected one has. Then, the selected route is returned to the calling process. By this way, we can select the most appropriate route from multiple routes.

5. State Processing in IP Packet Transformer

The role of IP is to transfer datagrams from the source node to the destination node. In a traditional manner, it is enough to transfer the IP packets in an as-is way. However, in these days, we have to "transform" the IP packets at the intermediate nodes, in order to execute wide variety of new functions. The security functions, such as authentication or encryption, are the typical examples. We need an implementation methodology, that is effective, flexible and extendable against such various "transformations."

In this section, we describe how the Serialized Data State Processing design applies to the generic IP Packet transformations. In this case, the process of state transition for each single transformation such as AH, ESP or IPComp (IP Compression) [13] is serialized. For example, the processing of IPsec in Linux can be integrated into the framework of IP Packet Transformer (i.e., XFRM), which is one of the instance or implementation methods of the Serialized Data State Processing design. We also show how we can naturally apply this methodology to the other functions, such as Mobile IP.

5.1 Stackable Destination and XFRM

New framework for IP packet processing has been introduced into Linux 2.5.x IPsec implementation, i.e., processing of Authentication Header [6] and/or Encapsulating Security Payload [7]. This methodology is called as "XFRM" or "Stackable Destination."

XFRM stands for transformer. The fundamental data structures of XFRM are `xfrm_policy{}` and `xfrm_state{}`. Each data structure represents the IPsec Policy (SP) and the Security Association (SA), respectively. `xfrm_policy{}` includes the Security Policy Database (SPD), and `xfrm_state{}` includes the Security Association Database (SAD). The `xfrm_state{}` is associated with `xfrm_policy{}` via `xfrm_tmpl{}`, that represents the template for packet transformation.

Stackable Destination (SD) is the infrastructure for packet transformation in the output path. It looks like a kind of linked list of `dst{}`. This list is created temporarily and cached according to the policy. `dst{}` has its own output method, `output`, and it transforms the packet in conjunction with `xfrm_state{}`, which represents the state of transformer.

The netlink [12] infrastructure is used as a native user interface to maintain SAD and SPD. In addition to this native interface, the standard `PF_KEY` [8] interface for SAD, and the `PF_KEY` KAME [5] extension for SPD are supported.

5.2 Address Family Independent XFRM Infrastructure

Since core functionality of the XFRM engine is common among address families (AF) (e.g. IPv6 and IPv4), AF independent XFRM infrastructure is introduced.

Instance of AF specific XFRM engine is instantiated by registering AF specific information tables, e.g. `xfrm_policy_afinfo{}` or `xfrm_state_afinfo{}`, to the core XFRM engine. Common variables are also passed via the tables.

5.3 Packet Processing Details

In this subsection, we describe the details of packet processing.

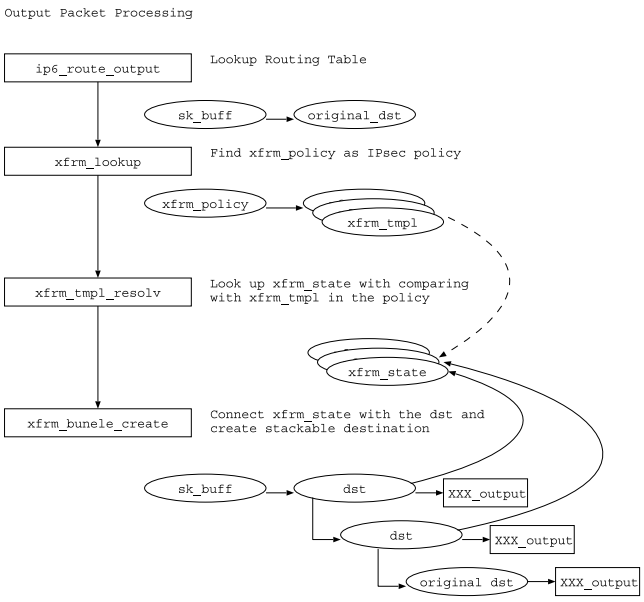


Fig. 11 IPsec output process with XFRM and SD.

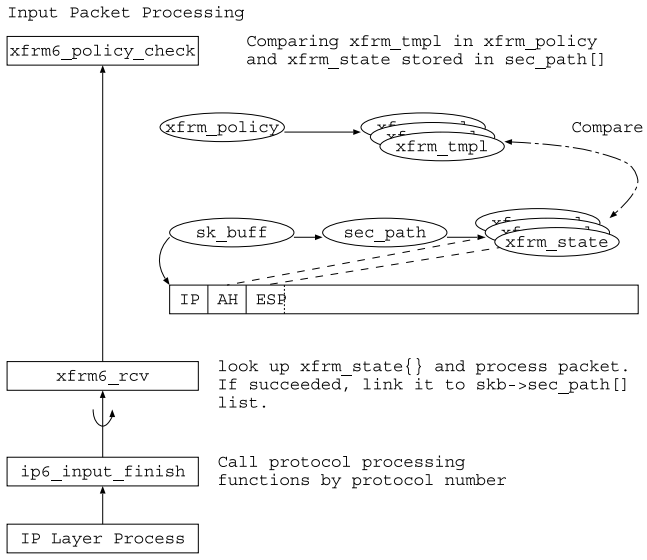


Fig. 12 IPsec input process with XFRM.

Table 2 Summary result of TAHI conformance test (Linux 2.5.58, %).

Test Series	Pass	Warn	Fail
ipsec	95	2	3
ipsec4	98	2	0
ipsec4-udp	96	4	0

5.3.1 Output Path

The output process of IPsec fully uses this architecture (Fig. 11). The following functions are sequentially called;

1. `xfrm_lookup()`,
2. `xfrm_tmpl_resolve()`,
3. `xfrm_bundle_create()` and `dst_output()`.

First, `xfrm_lookup()` looks up the `xfrm_policy{}` in SPD after the routing resolution. At this moment, the parameter `dst{}` in the stack points out the original `dst{}` structure. `xfrm_tmpl_resolve()` is called in `xfrm_lookup()` in order to resolve the `xfrm_tmpl{}` in `xfrm_policy{}` which represents how the packet is processed and finds the set of `xfrm_state{}` for it. This process corresponds to the looking up of IPsec SA (or IPsec SA bundle if multiple SA are needed) matched with the IPsec policy.

Then, `xfrm_bundle_create()` creates the Stackable Destination. This corresponds to creating of IPsec SA (or SA bundle).

Finally, `dst_output()` is called after the building up of the packet. Each output routine specified by the function pointer in the `dst{}` is called along with the chain of `dst{}` by popping up the `dst{}`. This pointer points out, for example, `esp6_output()`. The output function is able to use the `xfrm_state{}` from the `dst{}` pointer in `sk_buff{}`. As a result, the original `dst{}`'s output function is called and the packet is transmitted.

5.3.2 Input Path

The input process for IPsec is simpler than the output process (Fig. 12).

As all other extension header handlers and protocol handlers are registered with `inet6_protos[]` during the initialization phase, the processing routines for AH and ESP are also registered to `inet6_protos[]` during the initiation phase.

When a packet arrives at the input handler, the kernel parses the received packet from the head so as to call the appropriate handler based on the registration table information.

Each handler of IPsec protocol looks up `xfrm_state{}`. If the other function succeeds, the `xfrm_state{}` pointer is kept in `sec_path{}`, which is in `sk_buff{}`. Here, `sk_buff{}` contains the packet itself.

Finally, `xfrm_policy_check()` is called at the entrance of upper layer process. In `xfrm_policy_check()`, the kernel compares `xfrm_tmpl{}` in `xfrm_policy{}` and `xfrm_state{}` in `sec_path{}`, so as to determine if the received packet is allowed for delivery.

5.4 Conformance Test Results

On 24th April, 2003, Tom Lendacky reported test results of Linux 2.5.x IPsec to netdev mailing list. The result is shown in Table 2.

5.5 Application to Mobile IP

In the previous subsection, we showed that the XFRM infrastructure and the Stackable Destination are the promising implementation schemes for IPsec. In this subsection, we describe the feasibility of this implementation scheme to

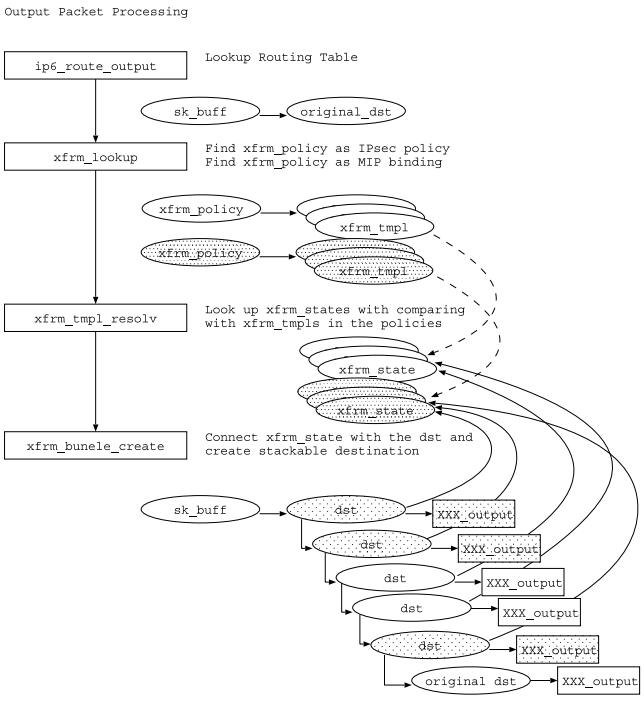


Fig. 13 Mobile IP using XFRM/Stackable destination scheme.

Mobile IPv6 [4], as well.

Figure 13 shows the operational diagram for Mobile IP implementation using XFRM/Stackable Destination infrastructure.

In this implementation, we introduce another set of policies for Mobile IP. These policies are controlled by the mobility daemon in user space based on the Binding Update information. The policy describes the type of transformation (e.g. appending Home Address Option in the destination options header) and the corresponding binding data (i.e. Care-of Address and Home Address).

The `xfrm_bundle_create()` compiles multiple templates into the single Stackable Destination, taking into account the flow of the process. When we assume the case where AH, ESP and Mobile IP co-exist in the header, the Stackable Destination should be constructed as follows.

- Mobile-IP Dest1, which inserts Care-of Address into the packet.
- Mobile-IP Rthdr, which inserts routing header (type 2) into the packet.
- ESP, which encrypts the packet after the destination options header for Home-Address option.
- AH, which generates authentication data for the packet and insert it after the destination options header.
- Mobile-IP Dest2, which swaps Care-of Address and Home Address in the packet.

6. Conclusion

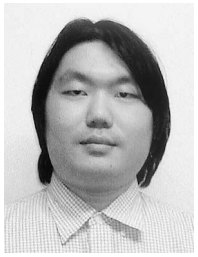
In this paper, we have introduced the Serialized Data State Processing design into the implementation of various IPv6 functions for legacy Linux system. The Serialized Data State Processing approach is a kind of combination among the simple self-managed data object, where one or more data objects are combined by directed links and the set of objects working all together. When we define the objects and their topology appropriately, we can eliminate the unessential (external) referencing against the other objects, while maintaining the object independency. In particular, when we can define the linear topology, we can easily eliminate the conflicts of resource dependency which frequently causes the deal locks.

First, we described the management structure of Neighbor Cache Entry. Second, we described the management structure of default router list, so that the processing of default router list is the same procedure for the normal routes. Third, we described the XFRM based implementation for IPsec and Mobile IP. Here, the XFRM is an instance of the proposed Serialized Data State Processing design.

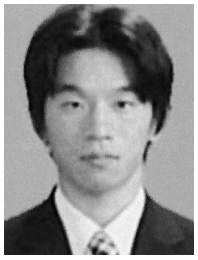
We can show that the proposed design is simple and extendable. The implementation described in this paper has been integrated into the globally available Linux distribution, as the USAGI Project IPv6 stack.

References

[1] 6bone, 6bone Home Page, http://www.6bone.net
[2] S. Deering and R. Hinden, "Internet protocol, version 6 (IPv6) specification," RFC2460, Dec. 1998.
[3] R. Drave and R. Hinden, "Default router preferences, more-specific routes, and load sharing," IETF Internet-Draft, June 2002.
[4] D. Johnson, C. Perkins, and J. Arkko, "Mobility support in IPv6," IETF Internet-Draft, June 2003.
[5] KAME Project, "KAME project Web page," http://www. kame.net
[6] S. Kent and R. Atkinson, "IP authentication header," RFC2402, Nov. 1998.
[7] S. Kent and R. Atkinson, "IP encapsulating security payload (ESP)," RFC2406, Nov. 1998.
[8] D. McDonald, C. Metz, and B. Phan, "PF_KEY key management API, version 2," RFC2367, July 1998.
[9] Microsoft Corporation, Windows XP Home Page, http://www. microsoft.com/windowsxp/
[10] T. Narten, E. Nordmark, and W. Simpson, "Neighbor discovery for IP version 6 (IPv6)," RFC2461, Dec. 1998.
[11] J. Postel, "Internet protocol," STD0005, Sept. 1981.
[12] J.H. Salim, H. Khosravim, A. Kleen, and A. Kuznetsov, "Netlink as an IP services protocol," RFC 3549, July 2003.
[13] A. Shacham, B. Monsour, R. Pereira, and M. Thomas, "IP payload compression protocol (IPComp)," RFC3173, Sept. 2001.
[14] K. Sklower, "A tree-based packet routing table for Berkeley Unix," in USENIX Winter, pp.93–104, Dallas, TX, 1991.
[15] Sun Microsystems, Inc., Solaris Operating System, http://wwws.sun.com/software/solaris/
[16] TAHI Project, Test and Verification for IPv6, http://www.tahi.org
[17] The FreeBSD Project, The FreeBSD Project, http://www.freebsd.org
[18] The Internet Society, Internet Engineering Task Force, http://www.ietf.org
[19] S. Thomson and T. Narten, "IPv6 stateless address autoconfiguration," RFC2462, Dec. 1998.
[20] USAGI Project, USAGI Project Web Page, http://www.linux-ipv6.org

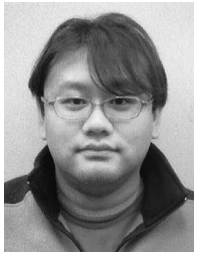


Hideaki Yoshifuji was born in Tokyo, Japan. He received the B.Eng., and M. of Information Sciences from Tohoku University, Sendai, Japan, in 1999 and 2001, respectively. He works for USAGI Project as core member since its establishment in 2000. Now he is a Ph.D. candidate at the University of Tokyo. He is one of Linux co-maintainers of networking area, since 2003.



Kazunori Miyazawa received B and M from Meiji University in 1997 and 1999, respectively. He joined Yokogawa Electric Corporation. He joined USAGI Project since 2000.

Masahide Nakamura received the B.E. and M.E. from Keio University in 1998 and 2000, respectively. He joined Hitachi, Ltd. in 2001, and then he was transferred to Hitachi Communication Technologies, Ltd. in 2002 when it was established. He works for USAGI Project as core member since 2002.



Yuji Sekiya was received B.E. from Kyoto University in 1997 and received M.E. from Keio University in 1999. He belongs to Information Technology Centre in the University of Tokyo since Oct. 2002. His major research topics are IPv6 and DNS. He works for USAGI Project as core member since its establishment.

Hiroshi Esaki received the B.E. and M.E. degrees from Kyushu University, Fukuoka, Japan, in 1985 and 1987, respectively. And, he received Ph.D. from University of Tokyo, Japan, in 1998. In 1987, he joined Research and Development Center, Toshiba Corporation, where he was engaged in the research of ATM systems. From 1998, he works for University of Tokyo as an associate professor, and works for WIDE project as a board member. He was at Bellcore in New Jersey (USA) as a residential researcher from 1990 to 1991, and was engaged in the research on high speed computer communications. From 1994 to 1996, he was at CTR (Center for Telecommunications Research) of Columbia University in New York (USA) as a visiting scholar. He is currently interested in a high speed internet architecture, including MPLS technology, a mobile computing, and IPv6.

Jun Murai is Professor, Faculty of Environmental Information, Keio University. Born in March 1955 in Tokyo. Graduated Keio University in 1979, Department of Mathematics, Faculty of Science and Technology, MS for Computer Science from Keio University in 1981, received Ph.D. in Computer Science, Keio University, 1987. Director, Keio Research Institute at SFC. The President of Japan Network Information Centre (JPNIC). He is appointed as one of the advisory Member of IT Strategy Headquarters establised within the Cabinet since August 2000. Adjunct Professor at Institute of Advanced Studies, United Nations University. He also teaches at Tokyo University of Art and Music. Specialized in computer science, computer network and computer communication. His recent publications include "Internet II," Iwanami Shoten, Publishers, July 1998, "Evolution and Revolution of the Internet in Japan," Proc. of CyberJapan: Technology, Policy Society Symposium, The Library of Congress, May, 1996.