Erlang 並行システムの SSH 分散トランスポート

力武 健次 [†]　中尾 康二 [†◇]

[†] 独立行政法人 情報通信研究機構 インシデント対策グループ
[◇] KDDI 株式会社 情報セキュリティフェロー

大規模ネットワークアプリケーションを実現する環境として並行分散システム Erlang が普及しつつある．しかし，Erlang の rpc モジュールによる分散プロセス実行環境はノード間の P2P 通信とポートマッピング機構に依存しており，広域インターネットではセキュリティ攻撃を受けやすい．本論文では，Secure Shell (SSH) によるノード間接続に基づく Erlang の分散プロセス実行環境を提案し，プロトタイプ実装による実験結果を評価する．

SSH Distribution Transport on Erlang Concurrent System

Kenji RIKITAKE[†], Koji NAKAO[†◇]

[†] Network Security Incident Response Group (NSIRG), NICT, Japan
[◇] Information Security Fellow, KDDI Corporation
Contact email: rikitake@nict.go.jp

The Erlang distributed concurrent system has been popular for a large-scale network application development. The distributed process execution environment of Erlang rpc module depends on point-to-point communication between the nodes and the arbitrary port-mapping mechanism, and is prone to security attacks on a wide-area Internet. In this paper, we propose a distributed process execution environment of Erlang based on Secure Shell (SSH) connections between the nodes, and evaluate the results of experiments with a prototype implementation.

1 Introduction

Web computing systems become more larger and complex every day, following the increase of users and scale of the communities over Internet. The implementation methodologies of such large-scale systems constantly change as new programming languages and systems emerge, notably from a centralized high-performance host to a cluster of synchronized distributed computers. For pursuing the overall processing performance, deliberate introduction of asynchronized system components loosely-coupled and allowing limited inconsistency with each other, has been gaining momentum among Web system designers.

The asynchronization of system components, to relax the condition of atomic full synchronization to partial synchronization within a set of multiple components, is ongoing in widely distributed computing environments, due to restriction of computing performance for each machine. The asynchronization is also needed to meet the requirement of geographically distributed systems to mitigate the risk of down time in clusters of hosts forming a computing cloud.

Even in the database applications where traditionally atomic consistency was on the top priority, other properties such as availability and partition tolerance are often on the higher priorities [1], also called with the property name *eventually consistent* [2, 3]. Web cache systems and Domain Name System (DNS) are typical examples of the eventually consistent systems.

To efficiently operate eventually consistent systems in a cloud without losing availability affecting the usability of the systems, a programming system with fault tolerance and concurrency-oriented properties is essential and critical. The *Erlang* programming language and the OTP (Open Telecommunication Platform) system [4] has been gaining popularity among Web programmers to provide the concurrency and fault tolerance. Erlang especially fits well with applications of massive simultaneous connections from the clients of short-term

processing, and has been a de-facto platform for ejabberd [5] Jabber/XMPP protocol [6]. Tsung [7] is another usage example of Erlang as a benchmarking tool to measure server response against large-scale simultaneous connections over XMPP, HTTP and other protocols.

The communication security of Erlang virtual machine (BEAM) is weak and insufficient for protecting possible attacks over Internet. We have presented an article of Erlang features and security weaknesses in a CSS2008 Symposium presentation [8]. Even in Erlang/OTP R13B01 Release as of June 2009[*1], the well-established remote procedure call (RPC) module of OTP, named *rpc*, does not include encryption functionality yet.

In this paper, we report a set of preliminary results, for applying OTP modules for SSH to implement a distributed process execution environment. The rest of the paper is organized as follows. In Section 2, we explain the related works of securing Erlang RPC. In Section 3, we explain the SSH and the underlying protocols, then we investigate how they can be applicable to securing Erlang RPC. In Section 4, we evaluate a prototype implementation of Erlang secure RPC over SSH. And in Section 5, we conclude this paper and propose the future direction of our research.

2 Related Works

We are not aware of previous studies and implementations of Erlang secure or encrypted RPC over SSH. We recognize, however, that Erlang/OTP itself has a sufficient set of cryptographic components to perform secure RPC, though it does not have a standard framework yet.

Erlang/OTP *crypto* module is a set of cryptographic protocols based on OpenSSL [9] library and the linked-in driver[*2] to allow up to 16 simultaneous cryptographic I/O operation. Erlang/OTP also has the native modules of Secure Socket Layer / Transport Layer Security (SSL/TLS) [10] and Secure Shell (SSH) libraries as *ssl* and *ssh* modules, respectively.

[*1] In this paper, we refer to the R13B01 Release simply as Erlang/OTP, unless otherwise noted.

[*2] Linked-in drivers of Erlang consist of registered shared libraries running in the same OS process for efficient execution. When it fails, the entire BEAM will crash.

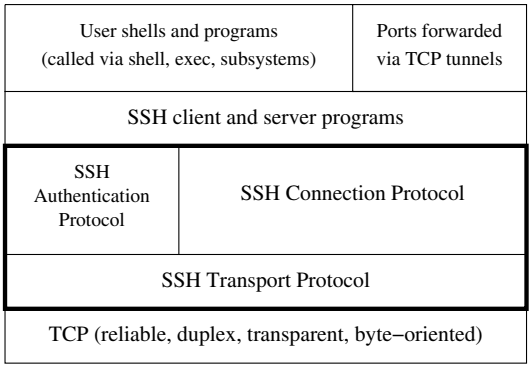

Fig. 1 SSH protocol architecture.

Erlang/OTP *ssl* module includes the distribution module called *inet_ssl_dist* [11], which mandates SSL encryption and authentication between BEAMs, based on shared certificate files. This module, however, does not coexist with non-encrypted *rpc* distribution module in the same BEAM, and the module does not encrypt communication between the port-mapping *epmd* daemons either. We will explain the details of *rpc* module in Section 4.

Erlang/OTP *ssh* module includes remote Erlang Shell execution for the read-eval-print loop (REPL) from a remote client over SSH. This REPL support is, however, limited for interactive use and is not suitable for RPC as is.

Secure Distributed Erlang (SDIST) [12] is a third-party Erlang module to provide basic functionality of RPC on the remote BEAM, such as spawning an Erlang process, invoking a module/function, and sending a message to an active process. SDIST uses SASL [13] authentication and provides support mechanism for TLS, and the access control lists for allowing and denying functions and modules to execute.

3 SSH protocol architecture

Secure Shell (SSH) [14][*3] is a set of protocol layers defined by Ylonen and Lonvick [15, 16].

SSH has a layered protocol structure over TCP as follows, also shown in Fig. 1:

[*3] In this paper, we discuss solely SSH Version 2 protocol, unless otherwise noted.

- SSH Transport Protocol [17], which handles server authentication, data compression, and other algorithm negotiation;
- SSH Authentication Protocol [18], which handles user and client authentication details; and
- SSH Connection Protocol [19], which handles channel multiplexing, out-of-band signals, and other functions required for terminal emulation and forwarding communication to/from other programs.

SSH Connection Protocol defines the bidirectional path for a remote execution as a *channel*. A connection between a SSH server and client can handle multiple channels. For running an arbitrary remote program, the protocol has three ways to connect:

- shell: invoking the default account shell for the authenticated user;
- exec: giving an arbitrary command string and waiting for the completion of execution under the user's privilege; and
- subsystem: invoking a program with preassigned abstract name.

For example, OpenSSH [20] has two file transfer mechanisms as follows:

- SCP, which uses an exec mechanism to connect the file transfer agents; and
- SFTP, which uses a dedicated subsystem to connect the client and the server.

The protocol also allows forwarding of TCP connections, X11 protocol of the X Window System, and SSH authentication agents.

4 RPC over Erlang ssh module

4.1 Design principles

Using RPC over Erlang/OTP *rpc* module, each BEAM instance, or node, establishes fully-connected mesh network between the other nodes, asking the IP address and port numbers of them through the control daemon *epmd*. While this approach allows a flat view of Erlang processes and resources among the mesh network, it also has the following problems:

- Erlang remote function calls via *rpc* module is not authenticated and fully trusted;
- *epmd* has no mechanism to encrypt the communication with each other, so is vulnerable to forgery attacks; and
- authentication between Erlang nodes is weak based on shared cookies.

Forming a virtual private network (VPN) is practically the only feasible way for protecting such a mesh network. Establishing a VPN requires an additional set-up procedure and is not always a practical solution. For example, we considered using IPsec for protecting the communication between Erlang nodes and *epmd* daemons, but we found out imposing a security policy with selectively encrypting TCP-based communication by the port numbers was impossible with freely-available policy managers such as racoon2 [21].

We decided to introduce a new RPC based on the Erlang/OTP *ssh* module and the Erlang programming language, instead of modifying the existing *rpc* module, for the following reasons:

- many existing applications, such as Mnesia [22] distributed database management system of Erlang/OTP, have already been dependent on the *rpc* module, and changing the semantics will break them;
- operating everything needed for SSH protocol handling within Erlang/OTP will make the RPC setup much easier than installing external programs written in other languages; and
- SSH public key management and the access control through the firewall routers and hosts are trivial tasks and will not impose new burdens for system administrators.

Our goal for this prototype implementation was writing a SSH server code to do the following:

- executing an arbitrary Erlang function in a module with arguments, given in a form of Erlang tuple as {`mfa`, {`module`, `function`, `argument_list`}};[*4]
- replying the execution result and the error status as an Erlang tuple immediately after each execution request; and

[*4] The atom `mfa` stands for "module, function, and arguments".

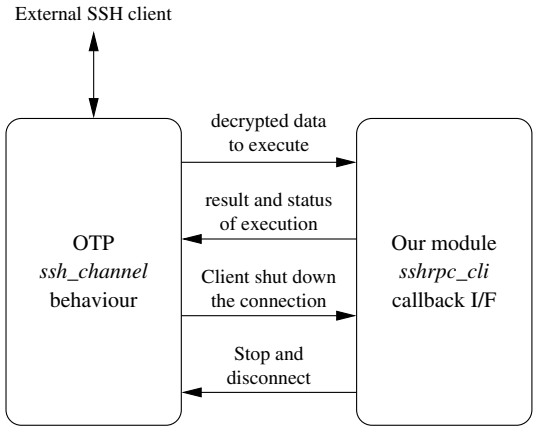

Fig. 2 SSH RPC server built on top of Erlang/OTP *ssh_channel* behaviour.

- automatic authentication between the RPC clients and servers which does not require intervention by a set of pre-defined keys.

4.2 SSH Implementation on Erlang/OTP

SSH protocol handling on Erlang/OTP is implemented as a *behaviour* [*5] called *ssh_channel*. Each behaviour is an extraction of generic process work and error handling, and can be built into user modules as `-behaviour(ssh_channel)`.[*6] Programmers using the behaviour have to develop the callback module and provide necessary callback functions specified in each behaviour. Behaviours provided in Erlang/OTP include those of client-server programming, finite state machines, and event handlers, which help the programmers to focus on their specific works and to share a common programming style [23, Chapter 12].

We implemented our RPC module *sshrpc_cli* as an *ssh_channel* behaviour, as a part of module called back from the SSH server function `ssh:daemon/3`.[*7] Figure 2 shows how our module working together with the *ssh_channel* behaviour.

In Erlang/OTP *ssh* module source code,[*8] we found an example callback interface module for providing I/O to the Erlang shell program called *ssh_cli*. Our *sshrpc_cli* module was a direct modification of *ssh_cli*, to provide the limited function as follows:

- when called as an SSH shell, the program returns the execution result of processing request, both the request and the result in Erlang external term format [24] for optimizing the content size;[*9] and
- when called by SSH exec command to process arbitrary string as the execution request, the program returns the execution result in Erlang external term format.

We also modified the Erlang/OTP SSH and cryptographic code to add the 128-bit AES Cipher Block Chaining (CBC) mode encryption handling capability [25] as follows, to satisfy the recommendation of RFC4253 [17] Section 6.3: [*10]

- added `crypto:aes_cbc_ivec/1` as a new function to update initialization vectors (IVs) of AES;
- added necessary code to encryption and decryption functions of *ssh_transport* module; and
- fixed the bug of handling zero-length packets in `ssh_transport:unpack/3`.

4.3 Prototype evaluation results

We conducted speed evaluation of three types of the SSH server callback code, as specified in SSH Connection Protocol [19, Section 6]:

- *exec*: using "exec" command and literal string form for each RPC call, following the reply of the execution result, channel-specific reply, exit status, channel end-of-file (EOF), and closing the channel and SSH session;
- *exec-mod*: using "exec" command and literal string form for each RPC call, fol-

[*5] In this paper, the word *behaviour* is solely used to describe the formalized design patterns of processes in Erlang/OTP.

[*6] `-behavior(...)` is also accepted for those who want to stick with American English.

[*7] Erlang function has a notation of identify itself in a set of modules as `module:function/arity`, where the arity means the number of arguments of the function.

[*8] At the directory `lib/ssh/src`.

[*9] Erlang has Built-In Functions (BIFs) for converting between Erlang internal terms and external binaries (untyped memory bits and octets) as `term_to_binary/[1,2]` and `binary_to_term/1`.

[*10] RFC4344 [26] further recommends usage of Counter (CTR) mode of AES on SSH.

Execution time of each RPC call via SSH

(unit: [ms])	exec	exec-mod	shell
μ	412.706	106.889	8.097
σ	2.632	0.353	0.261

μ: mean value / σ: standard deviation

Node specification and test conditions:

- CPU: Intel Atom N270 (max 1.6GHz)
- clock: 800MHz, frequency lowered by FreeBSD *powerd*
- memory: 1Gbytes
- OS: FreeBSD 7.2-RELEASE
- The client and server hosts connected via 100BASE-TX LAN
- Used Erlang R13B01, native code support (HiPE) enabled for the BEAM compiler, but not enabled for the modified and added code. BEAM is SMP-enabled (2 schedulers/node).
- Each set of execution time measured with 11 iterative executions of `erlang:now/0` which returns the internal wall clock value, and recorded elapsed time on the server side by taking the difference between the 1st and 11th attempts. Conducted 10 sets for each configuration and measured the statistics.

Table 1 Execution time of RPC calls to *sshrpc_cli* module.

lowing the reply of the execution result, channel-specific-reply, channel EOF, and does *not* close the channel even after the completion of execution to make a persistent connection; and

- *shell*: using "shell" command to start up the link, receiving Erlang tuple in the external term format for each RPC call, following the reply of the result and the status code as a single Erlang tuple in the external term format, and repeats this without closing the channel.

We configured the RPC client and server systems to use SSH RSA-based public-key server and user authentication. Erlang/OTP has following limitations for SSH key management:

- only the 1st-found key in the user public key is valid (`ssh_file:lookup_user_key/2`); and
- private key for RSA user authentication cannot be encrypted by a user password (`ssh_file:read_private_key_v2/2`).

Table 1 shows the wall clock time results of iterative RPC calls between two nodes on different hosts. While the *exec* and *exec-mod* showed a poor performance less than 10 calls/second, *shell* showed the performance exceeding 100 calls/second and would be useful for the real-world application. The results suggest that bringing SSH sessions up and down consume a lot of time, presumably by performing per-session negotiation between SSH protocol layers. Keeping SSH sessions persistent as possible is the key to reduce the connection overhead. Also, reducing the SSH Connection Protocol messages, such as the exit status code [19, Section 6.10], helps speeding up the transaction.

5 Conclusion and Future works

In this paper, we described the current issues of Erlang/OTP RPC systems, and showed how SSH protocol could be applied to build a secure RPC subsystem between Erlang nodes. We conclude that SSH RPC of Erlang/OTP would be useful for the real-world application, according to the achieved performance of more than 100 calls per second.

Our future works will include further optimization and improvement of the SSH RPC modules as follows:

- making the whole RPC as an SSH subsystem so that the RPC can co-exist with existing Erlang shell execution methods;
- allowing multi-packet Erlang terms on the SSH connection channels, by introducing states of fragmented Erlang binaries;
- improving the transaction performance by further optimization of underlying protocols such as TCP, including introduction of a new transport such as SCTP;
- performance evaluation over a long-haul wide-area network, including over IPv6 networks; [*11] and
- implementing more complex RPC functions such as asynchronous calls, messages

[*11] Erlang/OTP *ssh* module is already capable to handle IPv6 connectivity, though the capability was not tested during the evaluation reported in this paper.

to running processes, and multicasting over multiple SSH-connected Erlang nodes.

Acknowledgments

The first author Rikitake would like to thank Dave Smith for offering us the SDIST specification. He also thanks Jayson Vantuyl, Kenneth Lundin, Witold Babyluk, and Richard Andrews for the valuable discussion on "Controlled interaction of two Erlang distributed networks" in the mailing list erlang-questions@erlang.org around August 25–28, 2009.

References

[1] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," SIGACT News, vol.33, no.2, pp.51–59, 2002. http://doi.acm.org/10.1145/564585.564601.

[2] W. Vogels, "Eventually Consistent," Queue, vol.6, no.6, pp.14–19, 2008. http://doi.acm.org/10.1145/1466443.1466448.

[3] D. Pritchett, "BASE: An Acid Alternative," Queue, vol.6, no.3, pp.48–55, 2008. http://doi.acm.org/10.1145/1394127.1394128.

[4] Ericsson AB, "Open Source Erlang." http://www.erlang.org/.

[5] "ejabberd community site." http://www.ejabberd.im/.

[6] XMPP Standards Foundation, "XMPP Standards Foundation Web Page." http://xmpp.org/.

[7] ProcessOne, "Tsung." http://www.process-one.net/en/tsung/.

[8] K. Rikitake and K. Nakao, "Application Security of Erlang Concurrent System," Proceedings of IPSJ Computer Security Symposium 2008 (CSS2008), IPSJ Symposium Series, vol.2008, pp.253–258, IPSJ, Oct. 2008.

[9] The OpenSSL Project, "OpenSSL." http://www.openssl.org/.

[10] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," August 2008. RFC5246.

[11] Ericsson AB, "Using SSL for Erlang Distribution." http://www.erlang.org/doc/apps/ssl/ssl_distribution.html.

[12] Dave Smith, "Securing Distributed Erlang." http://dizzyd.com/sdist.pdf.

[13] A. Melnikov and K. Zeilenga, "Simple Authentication and Security Layer (SASL)," June 2006. RFC4422.

[14] D.J. Barrett, R.E. Silvermann, and R.G. Byrnes, SSH, The Secure Shell: The Definitive Guide, 2nd ed., O'Reilly & Associates, 2005. ISBN-13: 978-0-596-00895-6.

[15] S. Lehtinen and C. Lonvick, "The Secure Shell (SSH) Protocol Assigned Numbers," January 2006. RFC4250.

[16] T. Ylonen and C. Lonvick (Editor), "The Secure Shell (SSH) Protocol Architecture," Jan. 2006. RFC4251.

[17] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Transport Layer Protocol," January 2006. RFC4253.

[18] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Authentication Protocol," January 2006. RFC4252.

[19] T. Ylonen and C. Lonvick, "The Secure Shell (SSH) Connection Protocol," January 2006. RFC4254.

[20] The OpenBSD Project, "OpenSSH." http://www.openssh.org/.

[21] The Racoon2 Project, "The Racoon2 Project Wiki." http://www.racoon2.wide.ad.jp/w/.

[22] Ericsson AB, "Mnesia Reference Manual." http://erlang.org/doc/apps/mnesia/index.html.

[23] F. Cesarini and S. Thompson, Erlang Programming, O'Reilly Media, 2009. ISBN-13: 978-0-596-51818-9.

[24] Ericsson AB, "External Term Format." a part of ERTS User's Guide, http://www.erlang.org/doc/apps/erts/erl_ext_dist.html.

[25] K. Rikitake, "Erlang R13B01 ssh and crypto module patch of aes128-cbc support on ssh." http://www.erlang.org/cgi-bin/ezmlm-cgi?3:mss:449:200908:jocpkoflfkoikpmnfcnj.

[26] M. Bellare, T. Kohno, and C. Namprempre, "The Secure Shell (SSH) Transport Layer Encryption Modes," January 2006. RFC4344.

Supplemental note for
SSH Distribution Transport on Erlang Concurrent System
by Kenji Rikitake 6-NOV-2009

This note includes supplemental information in my presentation of
26-OCT-2009.

* Related works (Section 2)

BERT-RPC (http://bert-rpc.org/) 1.0 has been released as an
inter-language data exchange format, largely based on Erlang external
term format.

* aes128-cbc modification (Section 4.2)

Patches described in Section 4.2 have been included in Erlang R13B02.

* Execution time (Section 4.3, Table 1)

ssh:daemon/3 enables TCP Nagle algorithm, which introduces intentional
delay on sending packets. When turning the Nagle algorithm off by

```
    {nodelay, true}
```

option in the ssh:daemon/3 option (at the 3rd argument list), the
execution time shortened to 4.2msec per call, or 238 calls/sec.

* Future works (Section 5)

The following features are already implemented by the time of
presentation:

  - making the server side as an SSH subsystem
  - allowing multi-SSH-packet terms (up to 2^32 bytes)

[End of memorandum]