

A Rendezvous in Network Experiment — Case Study of Kuroyuri

Ken-ichi Chinen, Toshiyuki Miyachi and Yoichi Shinoda
info@starbed.org

Internet Research Center, Japan Advanced Institute of Science and Technology, JAPAN.

Abstract - *Rendezvous among experiment entities appear in network experiments both implicitly and explicitly. When the driving system for a network experiment testbed supports such rendezvous, users are free to focus on the content of the experiments, without having to consider details of the rendezvous (conditions, timing, etc.) and their execution. This paper shows typical entity activities in network experiments, especially rendezvous, and presents our approach in the Kuroyuri network experiment driving program.*

I. INTRODUCTION

Since network experiments typically involve various equipment and many programs, a driving program is needed to coordinate the experiment's environment and facility. Programs run concurrently, but in most experiments, the action of some entities are often dependent on the actions of others. For example, clients expect their server to be running. The log collection and analysis start after the appearance of target phenomena. Thus, network experiments need a way to order and control their steps.

In this study, we categorize context activities (processes and threads) as forking, termination and rendezvous. Forking is the creator of context. It is called "fork" and "spawn" in many platforms. Network experiments consist of many contexts. Part of them fork from the OS over the entities. They fork several times according to their roles, creating parallel activities.

When the context achieves its role, it terminates. A iteration of forking and termination forms a serial activity. Since most experiments are sequences of contexts, serialization is indispensable in the driving of experiments, though it tends to be forgotten.

Rendezvous is a context waiting for another's actions. Network experiments generally involve parallel activities. Rendezvous appear in them as the synchronization of experiment steps. In dis-

tributed denied of service (DDoS) experiments, stress generators should send requests simultaneously. Such experiments require rendezvous support in the driving system.

There are traditional ways to make a rendezvous employ UNIX "signals" (c.f., `kill` and `signal`) and similar things; they can be used for distributed inter-process signaling by TCP or UDP. These methods are simple and easy to implement, so they are reasonable for small and/or simple experiments. However they are not suitable for complex and/or large experiments because the values with which the "signal" can be expressed are limited (typically to a half of dozen, while others are reserved for the system.) The mechanism which sends various values is required to have a generic purpose.

On the other hand, communication between entities is required if they are perform their functions. By sending common data, entities can work together, changing their behavior according to the sent parameters. Imagine that you want to randomly dispatch many targets to contexts on multiple entities. It is easier to send random order targets to the common program on entities than to make individual programs with random targets for each entity.

Since we view the signaling as a kind of message, we build the signaling over the message exchange mechanism be-

tween entities. Then, users can write a rendezvous like a message exchange. These message exchanges are designed to be entity-based. Because context-based exchange forces the mechanism of message exchanging into every context, we do not employ that. It is too much for little programs, like `ping`. It would require the modification of the program.

Furthermore, our driver program sends entities' behaviors over the message exchange mechanism at the beginning of the experiments. The program stores all node behaviors in one place. This enables the user to change behavior and its parameters by the modifying one file. Users should not distributes behaviors to nodes manually, since that often leads to confusion and mistakes.

In this paper we present the concept, design and implementation of rendezvous for network experiments in the Kuroyuri network experiment driving program. First, however, we will introduce message exchanging, the handling of context activities, and others' approaches as a base for the rendezvous.

II. CONCEPTS AND REQUIREMENTS

A. Programmable and Traceable

Different users want to run their experiments with different parameters, so the experiment driver must be programmable. Since the user will repeat the experiment many times, the programming has to be traceable. Thus,

a language processing system is recommended. Simple input recording systems

(screen, a history of shells, and some keyboard macro mechanisms) are

not suitable in such situations.

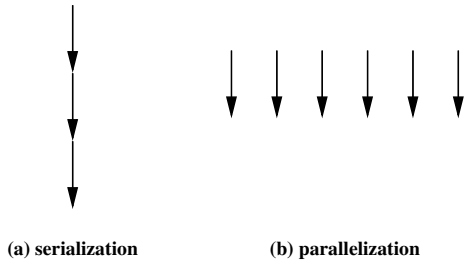


Fig. 1. Serialization and parallelization

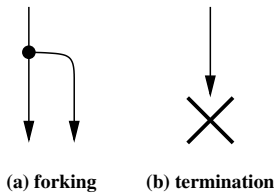


Fig. 2. Forking and termination — context activity primitives

B. Supporting of Context Activities

Program contexts of network experiments run both in series and in parallel (Fig. 1 .) Forking and termination are the context activity primitives (Fig. 2 .) Forking creates new contexts; it causes parallelization. Serialization is a combination of forking and terminations, as shown in Fig. 3. When many contexts are running, they need to rendezvous (Fig. 4 .) In micro view, the rendezvous is the waiting for a signal from other contexts. The driving program must support these context activities.

C. Message Exchanging

For convenience of writing, the driving program has to include features for message exchange between entities. When the program has such features built-in, the user can easily write a message passing procedure, as follows:

```
req = recv(X);
if(req equal "bird") {
  send(X, "eagle");
}
else
if(req equal "fish") {
  send(X, "tuna");
}
else {
  send(X, "what");
}
```

D. Rendezvous

As mentioned above, the rendezvous of experiment contexts is required. For example, here we show the gathering of service requests from clients to a server. Conceptually, it is written as follows:

```
wake clients after server up.
terminate server after clients down.
```

However this is an abstract description which is difficult to parse. A more specific procedural description is recommended:

```
while(1) {
  if(is-serverup())
    break;
}
wake-client1();
wake-client2();
...

while(1) {
  if(is-client1down()
    && is-client2down()
    && ... ) {
    break;
  }
}
term-server();
```

Preparing the various individual checking functions (is-setup(), is-client1down() and others are shown in the sample) is difficult, and

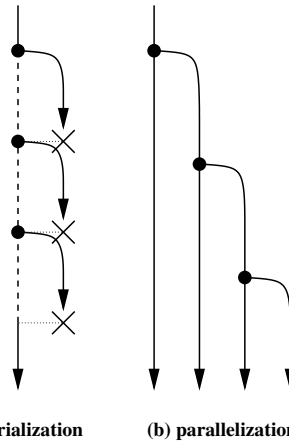


Fig. 3. Serialization and parallelization by forking and termination

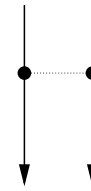


Fig. 4. Rendezvous

writing the waiting loop requires careful consideration. Users often write busy and/or worthless loops.

Thus, we propose an explicit rendezvous syntax to improve these issues. Using the model of a message base, it simplifies the description of the rendezvous. The following script equals the one shown above.

```
sync {
  msgmatch server "up"
}
send client1 "wake"
send client2 "wake"
...

sync {
  msgmatch client1 "down"
  msgmatch client2 "down"
  ...
}
send server "term"
```

By writing conditions, the user can use rendezvous easily, without the complexity and difficulty of writing procedural descriptions.

III. RELATED WORKS

A. Shells and Similar Things

Typical shell evaluators do not have features for message exchange over networks, nor are rendezvous found in them. To archive rendezvous on shells, users have to build programs which satisfy their specifications. Although message exchanging and rendezvous can be implemented in this way, there will be some problems because external programs often cause delay and because the management of multiple connections between entities heavily loads to the system. To achieve a higher performance, user should make those features built-in. However, it will still be heavy, and its benefit is small. Furthermore, users have to distribute the scripts manually and/or

semi-manually by the program itself.

Enhancements of shells, rsh [1] and ssh [2], do not satisfy our requirements either. rsh and ssh enable the remote execution of program but they do not support message exchange and rendezvous.

Expect [3], a kind of script language, has features of message exchange and condition switching by message. It is better than typical shells. However, the language is not designed for rendezvous.

B. Elvin4

NetBed [4] employs Elvin4 [5] for its communication channels. Elvin4 is a general purpose framework for distributed inter-process communications. It is an application multicast and is implemented as a library with a program.

We suppose that users can write a rendezvous implicitly by the time base control mentioned in Ref. [4]. However a

Elvin4-based program cannot wait for an action which requires an uncertain amount of time. Experiment programs often take a longer time than the user expects. In addition, many users want to measure that time in their experiments. Users who know the waiting time are in the minority. Thus, time base control is weaker than Kuroyuri's explicit rendezvous by message; it can wait such actions by the specification of waiting for "done" message.

C. CORBA

CORBA [6] is a platform of distributed middle-ware. It enables the calling of distributed objects on remote hosts that a network experiment is using. In this case, the distributed objects are the target programs of experiments.

However, using CORBA in a network experiment testbed is not reason-

able. The target program has to be a part of the CORBA programs, which forces the re-implementation of targets. Since the behavior of most programs is often changed by re-implementation, this is an undesirable situation. Some programs are released by binary file. The user cannot modify programs in such cases.

CORBA script language [7] might be very convenient, if the targets are limited to programs under CORBA. But it has no statement of rendezvous, so users are required to write complex scripts.

D. Polygraph

A WWW proxy benchmarking system Polygraph [8] supports staging, a kind of rendezvous, but it is implicit. It does not allow the modification of rendezvous by user descriptions.

TABLE I
NODE PROPERTIES

property		description
major	minor	sample
evaluator	ipaddr port	10.0.0.3 4312
network interface	ipaddr macaddr name media	192.168.4.8 00:12:34:56:78:9A fxp0 fastethernet
message	- lastmsg	buffer for reading pointer for last mssage
scenario	-	node behavior

IV. DESIGN

Kuroyuri, our network experiment driving program, is an evaluator for network experiment description language. It satisfies our requirements in that it is both programmable and traceable.

A. Node Structure

Here, we define the term "node." It indicates a computer like PC/PDA among entities, not a router/switch. This is because routers and switches are out of the scope of this paper, and user programs cannot run on them.

Since nodes have many properties, the

evaluator stores them with the structure. The structure is separated into information about the evaluator, network interfaces, message control, and the scenario (TABLE I.) The scenario is a sequence of statements describing behavior of the node.

The following script is a sample of the definition of the node S. The evaluator of the nodes waits for access from a remote node with the IP address 10.0.0.3 and port 4312. This node prints "helo", sends the message "ready" to node C, and prints "done" when it running.

```
node S {
  agent ipaddr "10.0.0.3" \
    port "4312"
  netif media fastethernet \
    rname "fxp0" \
    ipaddr "192.168.4.8" \
    macaddr \
      "00:12:34:56:78:9A"
  scenario {
    print "helo"
    send C "ready"
    print "done"
  }
}
```

B. System Structures

Kuroyuri consists of two programs, master and slaves (Fig. 5.) The former runs

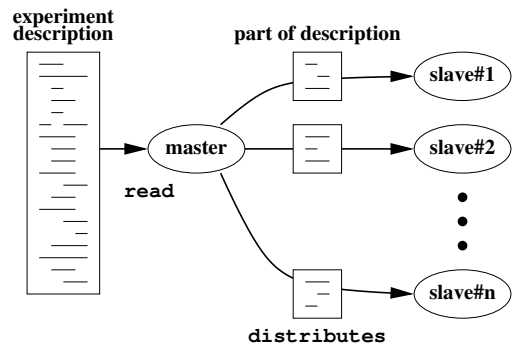


Fig. 5. Evaluators — master and slaves

on the node used to drive the experiments. The program, master, parses the user's description and compiles it to the inner-language. The program splits off part of the description for each node and sends them to the nodes. The slaves run on experiment nodes. The program evaluates the sent descriptions.

C. Context Birth and Dead

To form parallelization by forking, a statement `wake` is available. The statement `wakes` a specified program with arguments. Similarly, the statement `wakewait` is used for serialization. It pairs forking the program and waiting for its termination.

```
wake path arg0 arg1 ...
wakewait path arg1 arg1 ...
```

Statement `exit` terminates the context. The user can specify an exit code.

```
exit 79
```

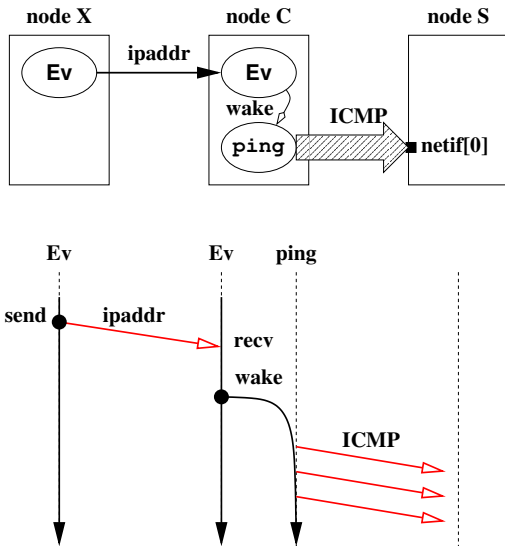


Fig. 6. IP Address Passing

E. Rendezvous

A rendezvous is a mixture of condition switching and message exchange. The following script shows a rendezvous with node `foo`:

D. Message Exchange

We design and the primitive statements `send` and `recv` for message exchange. Using them, scripts can exchange data and program parameters. The statement `send` means the sending of the specified message to the evaluator of the specified node.

```
send A "hello"
```

With the statement `recv`, the driving program receives a message from the specified evaluator and stores it. The storage position is specified by the name of the variable. The following script means "receive message from node B and store it to the variable `msg`."

```
recv B msg
```

Since condition switching by message contents occurs frequently in experiments, we introduced an optimized statement `msgswitch` for that purpose. The message exchange example in Section II.C is written as follows:

```
recv X req
msgswitch req {
  "bird" {
    send X "eagle"
  }
  "fish" {
    send X "tuna"
  }
  default {
    send X "what"
  }
}
```

Program Parameter Passing

The following script is an example of the passing of program parameters. It sends the IP address of node `S`'s first network interface to node `C`.

```
send C S.netif[0].ipaddr
```

The next example shows the sending of ICMP packet three times to a specified IP address by the program `ping`.

```
recv X target
wake "ping" "-c" "3" target
```

Fig. 6 depicts these behaviors. "Ev" in the figure indicates some evaluator.

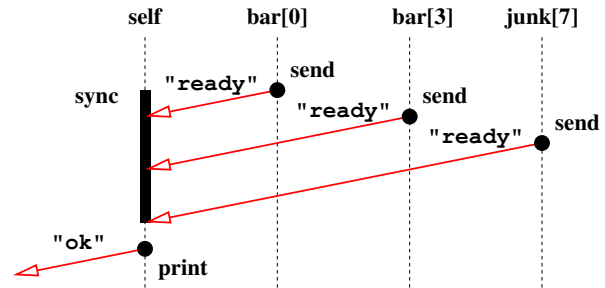


Fig. 7. Sync. Three Node Messages

`timeout` specifies the waiting time limitation in seconds.

Kuroyuri also supports rendezvous for multiple nodes. The next example shows a rendezvous with node `bar[0]`, `bar[3]` and `junk[7]`:

```
sync {
  msgmatch bar[0] "ready"
  msgmatch bar[3] "ready"
  msgmatch junk[7] "ready"
}
print "ok"
```

Fig. 7 shows the behavior of this rendezvous. The evaluator prints an "ok" message after receiving "ready" messages from the three nodes.

```
sync {
  msgmatch foo "ready"
  timeout 60
}
```

Block `sync` is a list of conditions for a rendezvous. It waits events until the conditions have been satisfied. Condition `msgmatch` is the comparison between the specified string and the message from the specified node. Condition

```

check-msgmatch(c)
{
  if(lastmessage(node(c)) eq "") {
    read-nextmessage(node(c));
  }
  if(lastmessage(node(c))
    eq target-string(c))
    return TRUE;
  else
    return FALSE;
}

foreach {
  if(check-timeout())
    break;
  nmatch=0;
  ncond=0;
  foreach c (<all conditions>) {
    if(is-msgmatch(c)) {
      ncond++;
      if(check-msgmatch(c)) {
        nmatch++;
      }
    }
  }
  if(nmatch==ncond)
    break;
  <sleep -- a breath for system>
}

```

```

node sv {
  ...
  scenario {
    ...
    wakewait "httpdctl" "start"
    send "ready"
    rcv cmd
    wakewait "httpdctl" "stop"
    ...
  }
}

nodeclass clC {
  ...
  scenario {
    ...
    rcv sname
    rcv wt
    wakewait "httpperf" "--num-calls"
      (tostring(wt)) "--rate" "1"
      "--server" sname
    send "done"
  }
}

n=4
nodeset cl class clC num n
...
scenario {
  ...
  rcv sv msg
  ...
  for(i=0;i<n;i++) {
    send cl[i] (haddr(sv.netif[0].ipaddr))
    send cl[i] (n-i)*30
    sleep 30
  }
  ...
  sync {
    multimsghmatch cl "done"
  }
  send sv "down"
  ...
}

```

Fig. 8. The pseudo code of rendezvous evaluation

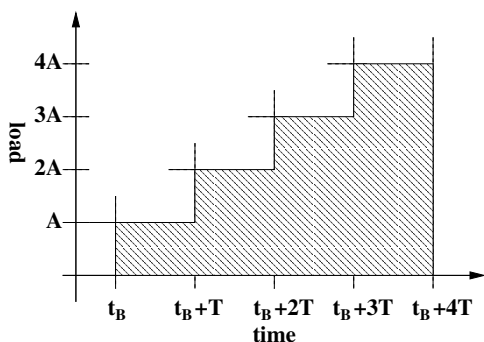


Fig. 9. Expected load

To avoid users' mistakes, we introduced the statement `multimsghmatch` as syntax sugar. It handles all nodes in a nodeset similar to `msgmatch`. When you have a nodeset `bar`, the rendezvous with all nodes of the nodeset is written as:

```

sync {
  multimsghmatch bar "ready"
}

```

Above script is equals to the shown below (N is the number of members of the nodeset `bar`):

```

sync {
  msgmatch bar[0] "ready"
  msgmatch bar[1] "ready"
  msgmatch bar[2] "ready"
  ...
  msgmatch bar[N] "ready"
}

```

F. Functions

Functions are useful for step gathering. The following short scripts are example of that:

```

func square(x) {
  x*x
}

func fact(x) {
  if(x<=1) {
    1
  }
  if(x>1) {
    x*fact(x-1)
  }
}

```

V. IMPLEMENTATION

We employ a kind of LISP as the evaluator engine because S-exp, a data type of

LISP, suits variable length, various data types and meta programming.

Kuroyuri is written in C language with approximately 50 thousand lines. We verify the running of Kuroyuri on FreeBSD, NetBSD, Linux and Solaris. TABLE II shows its detail.

A. Node

The evaluator handles the node structure as a primitive type, like integers and strings. When the evaluator runs as part of SpringOS, attributes of nodes are solved by asking the resource manager. The manager binds nodes to physical resources. See Ref. [9] for details.

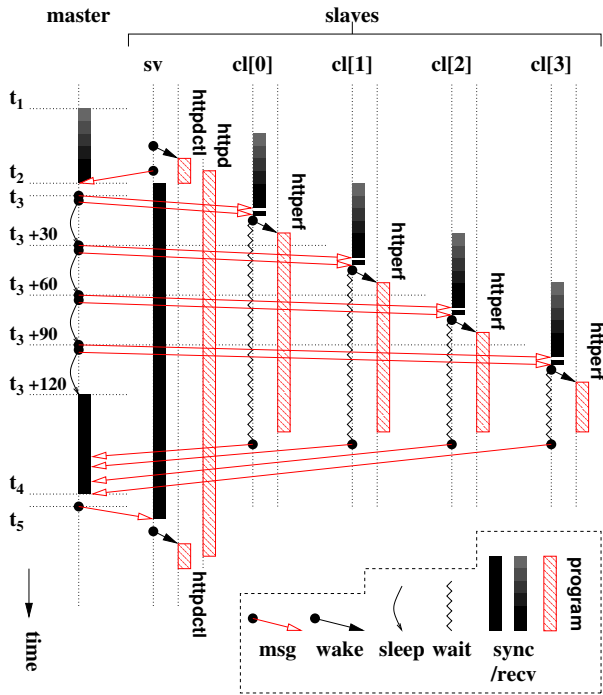


Fig. 11. The time chart of evaluation

B. Message Stream

The message exchange mechanism is built on a TCP connection using a BSD socket library. The evaluator handles the connection as a S-exp

stream. Since TCP is a byte stream, the handling includes buffering. This buffering is most complex part of the evaluator. `recv` is a simple reading from the S-exp stream. This S-exp reading enables variable length data handling. It can read integers, strings, symbols and lists. Structured data (including the descriptions of nodes) are read as lists. The subroutine for receiving stores the read data into the data area for streaming before returning it to its caller. This stored

TABLE II
VERIFIED PLATFORMS

OS type	version
FreeBSD	4.4
	5.0
NetBSD	1.6
	2.0
TurboLinux	7.0
	(kernel 2.4)
SunOS	5.9

data is used by the rendezvous. On the other hand, the output side of the connection is not buffered. `send` is a direct writing to the TCP connection.

C. Program Passing between Master and Slaves

The evaluation at the slave starts when the slave receives a message from the master. To the node, the evaluation of this message means the experiment. The message consists of the background and behavior of the node. The background is a set of node properties. The behavior is the script for the node. So, the double read-eval loop makes the slave. It is as simple as the following:

```

forever {
  background = read(<master>);
  eval(background);
  scenario = read(<master>);
  eval(scenario);
}

```

D. Rendezvous Handling

The rendezvous features depends on the message exchange module. When the evaluator meets a `sync` block, the program checks the satisfaction of each condition. If the timeout is satisfied, the pro-

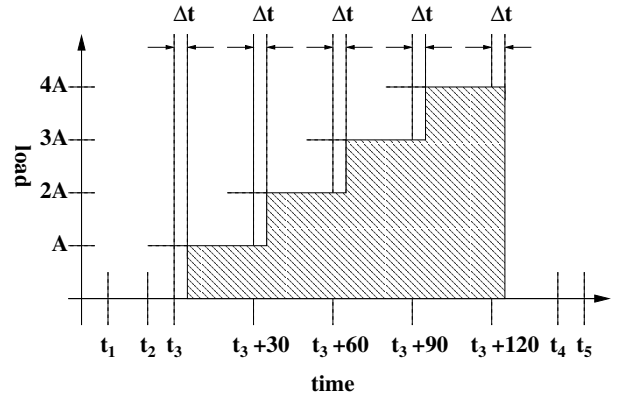


Fig. 12. Generated load

gram breaks the block. There are two phases for the checking of `msgmatch` conditions. When the stream receives messages, the evaluator compares the last message and the specified string. Otherwise, the evaluator reads the new message and compares the message and the string. This trick is necessary because the evaluator is looped to handle many streams in `sync`. If the program applies the reading to all streams at every repeat, the target message will pass by. Remember, the rendezvous is a kind of waiting. Since the passing by is an unexpected result, the program skips the reading when it has a message. Fig. 8 shows these steps. The sleep at the end is a tuning parameter that improve the response time of the node, because the loop without sleep probably into a "busy loop."

E. Other features

Functions are projected as special symbols with a lambda-function. The function `square`, mentioned above, is converted to the following:

```

(setq square (quote
  (lambda (x) (* x x))))

```

VI. EXAMPLE

We introduce a longer example to demonstrate Kuroyuri's ability. In this example, we try to generate stair load. Fig. 9 shows that with 4 steps. The graph raises unit load A per unit time T . Assume the target and its generator are HTTP[10] and `httperf`[11].

By the time-base rendezvous, user will

recognize the difficulty of starting at t_B because user does not know the booting time of the HTTP server. The user may be able to set that starting time from experience. Otherwise, he/she will probably sets a very long time.

Fig. 10 shows one of the solutions for this problem by Kuroyuri. Node `sv` and nodeset `cl` are defined. At `sv`, `httpd` starts and stops by `httpdctl`. The evaluator of `sv` sends the message "ready" after the start of the server. The server stops when the message arrives. Node `cl` receives two messages and uses them as the program parameters, the name of server and the number of requests. `httpperf` runs with these parameters. The evaluator sends the message "done" to the master.

The evaluator of the master waits for the message "ready" from node `sv`. `recv` is enough to receive the message in this case, because the sender and its message are clear. This receiving brings the above starting time t_B . Next, the master sends the IP address of `sv` and the length of time for the load. It is repeated with 30 seconds sleep, increasing the load every 30 seconds.

Clients messages rendezvous at the `sync` block, which absorbs jitters of client progress. After the collection of client terminations, the master terminates the server by sending of the message "down" to `sv`.

Strictly, we expect some errors in this experiment. Fig. 11 is the time chart of evaluators. There is a gap between the time the IP address is sent and the time `httpperf` starts, which is not observed at master. Fig. 12 depicts the gap Δt . However, the gap is constant, certainly. The interval of the step will remain constant. It satisfies the requirements, the shape of the load (Fig. 9.)

Finally, this example shows that Kuroyuri enables

- 1) Parameter passing by message exchange
- 2) Message-based rendezvous (explicit)
- 3) Time-base rendezvous (implicit)

VII. DISCUSSIONS

Although we have verified the running of the driving system on 250 or more nodes,

scalability is still a worrying point. Using some virtual technologies, the scale of network experiment testbeds will be thousands and more in the near future. We expect that handling experiments on that scale with the current approach will be difficult and complex. We should consider a new model (hierarchical management, self organization and others) for that situation.

VIII. CONCLUSION

To drive network experiments, users have to handle multiple contexts on multiple nodes. It also requires the synchronization contexts in complex experiments. We categorize context activities into forking, termination and rendezvous. The system which supports these activities can drive the experiments.

We designed the network experiment driving program Kuroyuri as the evaluator for network experiment description language. Experiments are driven by the parsing and evaluation of the program.

The special feature of Kuroyuri is a rendezvous based on the message exchange between nodes. The evaluator waits until the arrival of messages or timeout. Users can describe the synchronization by simple writing the conditions.

ACKNOWLEDGMENT

The authors would like to thank users of StarBED and SpringOS for their worthwhile questions and opinions. Parts of Kuroyuri's concepts and requirements are based on their comments and suggestions.

REFERENCES

- [1] B. Kantor. BSD Rlogin, RFC1282. December 1991.
- [2] T. Ylonen. SSH - secure login connections over the internet. In *the 6th USENIX Security Symposium*, p. 37, 1996.
- [3] Don Libes. The expect home page. <http://expect.nist.gov/>.
- [4] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet

Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI02*, pp. 255–270, Boston, MA, December 2002.

- [5] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content based routing with elvin. In *AUUG2K*, June 2000.
- [6] Object Management Group. *Common Object Request Broker Architecture: Core Specification v3.0*, March 2004. <http://www.omg.org/doc/formal/04-03-12.pdf>.
- [7] Object Management Group. *CORBA Scripting Language Specification v1.1*, February 2003. <http://www.omg.org/docs/formal/03-02-01.pdf>.
- [8] Measurement Factory. Web polygraph. <http://www.web-polygraph.org/>.
- [9] Toshiyuki Miyachi, Ken-ichi Chinen, and Yoichi Shinoda. Automatic configuration and execution of internet experiments on an actual node-based testbed. In *Tridentcom 2005*, pp. 274–282, February 2005.
- [10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, RFC2616. June 1999.

- [11] D. Mosberger and T. Jin. httpperf: A tool for measuring web server performance. *Performance Evaluation Review*, Vol. 26, No. 3, December 1998.

A AVAILABILITY

Kuroyuri is available at <http://www.starbed.org/>. The program is included in the suit of network experiment assistant tools, SpringOS. You can retrieve and use them freely.