FLoM: the *F*ree *Lo*ck *M*anager

Locking is evil, but if you need it, try to avoid the hell

Christian Ferrari, tiian@users.sourceforge.net, 2016

Abstract

This paper introduces FLoM, a free and open source software project that provides a tool for local and distributed process synchronization.
FLoM provides both a command line utility and a client library: the command line utility is intended for the system engineer while the client library allows the system programmer to integrate FLoM with its own custom software.
FLoM provides a rich and flexible semantic that simplify the implementation of complex synchronization scenarios.
Last but not least, the software implements an event-driven communication protocol to provide good scalability characteristics.

Introduction

FLoM is a free and open source software licensed under the terms of the GNU Public License and can be freely downloaded from GitHub and from SourceForge [1].
Process synchronization is discouraged in the implementation of scalable systems as explained in *The Reactive Manifesto* [2], but for many daily tasks it's helpful.
Sometimes, it is used to reduce the complexity of an algorithm: assuming that some data cannot be concurrently accessed simplifies code development and test.
Other times, process synchronization cannot be avoided because it's necessary to guarantee the logical correctness of an algorithm.
In the event that you decide to implement some process synchronization, FLoM will be an easy to use and flexible solution.
FLoM provides two fully integrated tools: a command line utility and a client library available for 5 different languages: C, C++, Java, PHP and Python.
The command line utility is designed for the system engineer that needs an easy tool to synchronize shell commands and shell scripts. Typical use cases are: avoiding the concurrent execution of two conflicting shell scripts, limiting the maximum number of shell scripts concurrently executed, assigning different processing directories to different shell scripts and so on…
The client library exposes a straightforward API that allows the system programmer and the software developer to synchronize some program portions. Typical use cases are: avoiding to read a file while another program is writing to it, obtaining a unique identifier or a unique timestamp while many concurrent programs are doing the same, distributing a set of items to distinct programs without conflicts and so on…
A distinctive feature of FLoM is the full integration between the command line utility and the multi-language client library. As an example, with FLoM you can synchronize a backup shell script, a batch Java program and a PHP web application.
All the above can be transparently used in completely different deployment environments: inside a single system or across a network of interconnected systems using the TCP/IP protocol. The support of IPv4, IPv6 and TLS/SSL allows a wide range of network deployments: LAN, MAN, WAN and Internet.

Architecture

FLoM implements a client/server architecture: the client is a light piece of code that sends requests and waits answers from the server. All the synchronization logic is managed by the server.
The server component, called `flom` daemon, must be activated using the command line `flom` utility.
The client component is provided in two different flavors: the command line `flom` utility and the `libflom` library.

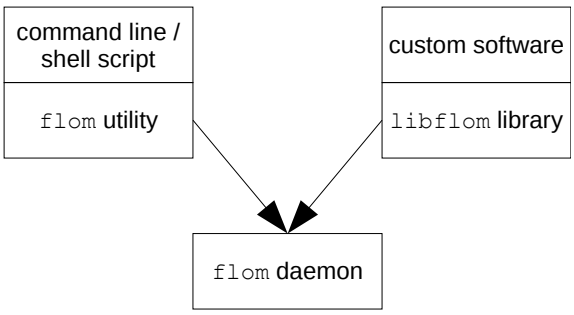

Figure 1: client/server architecture

All the components shown in Figure 1 can live in a single Linux system or in a distributed environment: the only requirement that must be met is network connectivity.

Implicit FLoM daemon (auto)start

FLoM is designed with the aim of reducing the complexity: in the easiest scenarios you don't have to explicitly start a `flom` daemon because it's au-

tomatically activated by the `flom` utility if necessary. The behavior can be described with this simple algorithm:

- at start up, the `flom` utility looks for an available `flom` daemon
- if the search ends with a positive result, the `flom` utility connects to the available `flom` daemon and uses it as the *synchronization server*
- if the search ends with a negative result, the `flom` utility activates a new `flom` daemon and uses it as the *synchronization server*

Explicit FLoM daemon start

Daemon "auto start" is a feature that can be easily disabled by the system engineer that wants to explicitly activate the `flom` daemon. As shown in section "Persistency and consistency", disabling "auto start" can be very useful for some use cases.

The FLoM client library (libflom)

`libflom` is a C library that provides a compact and easy to use API that can be used to synchronize custom developed programs using the FLoM model. Together with `libflom` for C, FLoM provides four further language bindings for: C++, Java, PHP and Python[1].
Software developed by the customer and linked to the `libflom` library cannot use `flom` daemon "autostart" feature and the `flom` daemon must be explicitly activated using the `flom` utility.

FLoM programming model

FLoM is developed using the C language and the `libflom` library exposes a standard ANSI C interface.
C++ and Java are truly object oriented programming languages, but the FLoM programming model is strictly based on the C native interface even for these languages. Nevertheless, integrating your own custom software with FLoM is really simple and straightforward.

C native interface

The basic usage requires only 1 "object" of type `flom_handle_t`:

```
flom_handle_t *handle;
```

and 4 functions:

```
flom_handle_new();
flom_handle_lock(handle);
flom_handle_unlock(handle);
flom_handle_delete(handle);
```

all the code between `flom_handle_lock` and `flom_handle_unlock` is synchronized.
All the configuration options can be specified inside the program using a
`flom_handle_set_`*`something`*[2]
function or outside the program using configuration files.
All the configuration options can be retrieved using a
`flom_handle_get_`*`something`*
function.
`flom_handle_set_`*`something`* is typically used after `flom_handle_new` and before `flom_handle_lock` to configure the behavior at run-time;
`flom_handle_get_`*`something`* can be used at any stage.
PHP and Python languages use a similar, C like syntax:

```
$handle = new flom_handle_t();
flom_handle_init($handle);
flom_handle_lock($handle);
flom_handle_unlock($handle);
flom_handle_clean($handle);

handle = flom_handle_t();
flom_handle_init(handle);
flom_handle_lock(handle);
flom_handle_unlock(handle);
flom_handle_clean(handle);
```

C++ interface

The basic usage requires only 1 object of class `FlomHandle`:

```
FlomHandle *handle;
```

and 4 methods:

```
handle = new FlomHandle();
handle->lock();
handle->unlock();
delete handle;
```

all the code between `handle->lock()` and `handle->unlock()` is synchronized.

1 The support for the C language is provided through FLoM native interface, the support for the C++ language is based on a header only wrapper of the C native interface. The Java language interface is based on native Java classes and a JNI mapping of the C native interface. PHP and Python interfaces are SWIG [3] based wrappers of the C native interface.

2 *`something`* must be replaced with the name of the interested option

All the functions available in the C native interface are mapped to methods in the C++ interface; configuration options can be changed with

`handle->set`*`Something`*

and inquired with

`handle->get`*`Something`*

Java language uses a similar syntax:

```
FlomHandle handle;
handle = new FlomHandle();
handle.lock();
handle.unlock();
handle.free();
```

Synchronization model

The synchronization model implemented by FLoM is based on the concepts of *resource* and *lock*.
The *resource* is the abstraction introduced by FLoM to support process synchronization: two or more processes synchronize themselves specifying the same *resource* and an *operation* ("lock", "unlock").
FLoM extensively uses the concept of *resource* to model process synchronization: the *client*, without distinction between `flom` utility and `libflom` library, asks to the *server* (`flom` daemon) to *lock* a specific *resource*.
The diagram in Figure 2 shows two processes that exclusively *lock* the *resource* "/foo/bar".

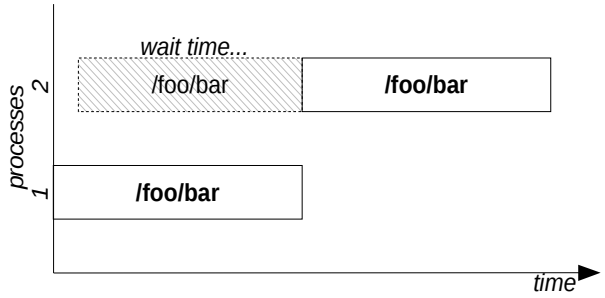

Figure 2: exclusive lock example

FLoM does not support only "exclusive locking", but implements the same *lock modes* introduced by the OpenVMS Distributed Lock Manager [4]: "null", "concurrent read", "concurrent write", "protected read" (*share lock*), "protected write" (*update lock*) and "exclusive" (*exclusive lock*). Table 1 represents the truth table and describes the lock modes behavior.

Mode	N	CR	CW	PR	PW	EX
N	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**
CR	**Y**	**Y**	**Y**	**Y**	**Y**	N
CW	**Y**	**Y**	**Y**	N	N	N
PR	**Y**	**Y**	N	**Y**	N	N
PW	**Y**	**Y**	N	N	N	N
EX	**Y**	**N**	N	N	N	N

Table 1: truth table for lock modes

The example in Figure 3 shows three processes:

1. locks resource "apple" using mode "protected read" (first process)
2. locks resource "apple" using mode "protected read" (second process)
3. locks resource "apple" using mode "exclusive" (last process)

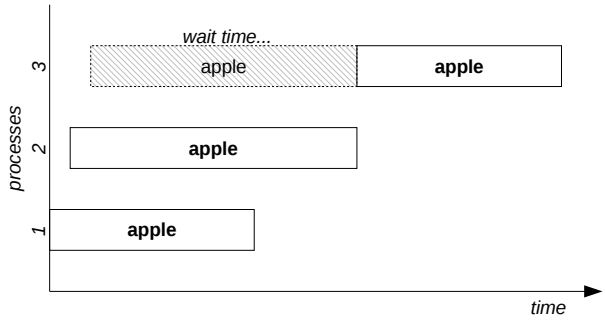

Figure 3: mixed lock mode example

Looking at the truth table we can see that "protected read" is compatible with "protected read", but "exclusive" is not compatible with "protected read"; the practical result is the concurrent execution of the first two processes and the delay of the third one.
A *resource* is completely specified by its name and the scope of a *resource name* is limited to a single `flom` daemon. Inside a `flom` daemon, a *resource name* uniquely identify a resource.
FLoM provides different type of resources to support a wide range of use cases: "simple", "numeric", "set", "hierarchical", "sequence" and "timestamp". As explained in the next paragraphs, some resource types support only a subset of the *lock modes* described in Table 1.
The type of a resource is automatically inferred from its name: there's no need to specify the type with an attribute. The syntax of resource names is based on regular expressions [5].

Simple resources

Simple resources are probably the easiest and most common resources to use; they supports all the *lock modes*.

Example: *command1* and *command2* can be synchronized to exclusively use resource *R1* with these commands:

```
flom -r R1 -l EX -- command1
flom -r R1 -l EX -- command2
```

To specify *protected read lock mode for command2*, use this command line:

```
flom -r R1 -l PR -- command2
```

To reproduce the scenario depicted in Figure 3, use these commands:

```
flom -r apple -l PR -- command1
flom -r apple -l PR -- command2
flom -r apple -l EX -- command3
```

Numeric resources

Numeric resources are useful to model synchronization processes related to multiple producers and/or multiple consumers; *numeric resources* can be used to implement workload capping. The only supported *lock mode* is "exclusive".

Example: to allow a maximum number of 2 concurrent executions of *command*, use a *numeric resource* as below:

```
flom -r limit[2] -- command
```

Independently from the number of different command lines, as shown in Figure 4, a maximum number of 2 *command* will be concurrently executed.

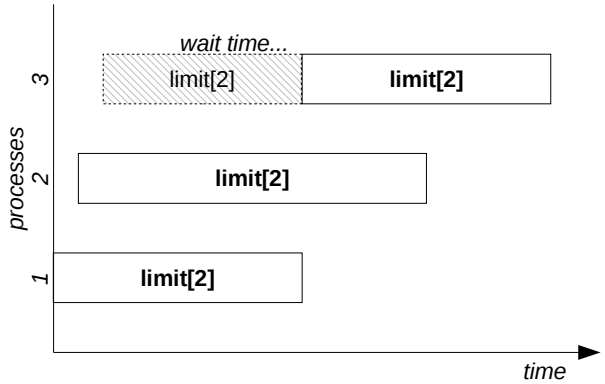

Figure 4: numeric resource example

Resource sets

Resource sets can be used to model synchronization processes with a finite set of distinct resources that must be assigned to the requesters: every requester can lock only one resource at a time. This type can be used to implement a simple *round robin scheduler*. The only supported *lock mode* is "exclusive".

Example: the *dosomething* command processes the content of a directory passed as first argument, 4 directories (cyan, magenta, yellow, black) must be continuously processed, but only one *dosomething* command at a time can work on a directory. The following command lines:

```
flom -r 'cyan.magenta.yellow.black'
-- dosomething
flom -r 'cyan.magenta.yellow.black'
-- dosomething
flom -r 'cyan.magenta.yellow.black'
-- dosomething
flom -r 'cyan.magenta.yellow.black'
-- dosomething
```

produce the same result of these ones:

```
dosomething cyan
dosomething magenta
dosomething yellow
dosomething black
```

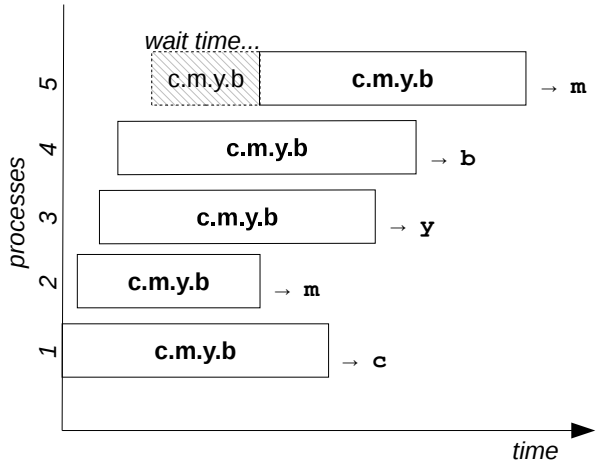

Figure 5: resource set example

Figure 5 shows an example with a resource set of 4 elements: the fifth process has to wait the completion of the second process and gets the same resource: "m".

Hierarchical resources

Hierarchical resources main usage is related to file system resources (files and directories); they support all the lock modes. *Hierarchical resources* support lock propagation: a lock at some level, propagates to all the underlying levels.

Even if the *hierarchical resources* natively model file system resources, they can be used to model any abstract hierarchical structure because there's no necessity to have the corresponding file system resources (FLoM resources are abstract).

Example: *command1* needs *protected read* access to */foo/bar* and *command2* needs *exclusive* access to */foo/bar/apple*:

```
flom -r /foo/bar -l PR -- command1
flom -r /foo/bar/apple -l EX --
command2
```

In the above example, *command2* will be executed after *command1* end because the *exclusive* lock at the */foo/bar/apple* level is not compatible with a *protected read* lock at the */foo/bar* level (see Figure 6).

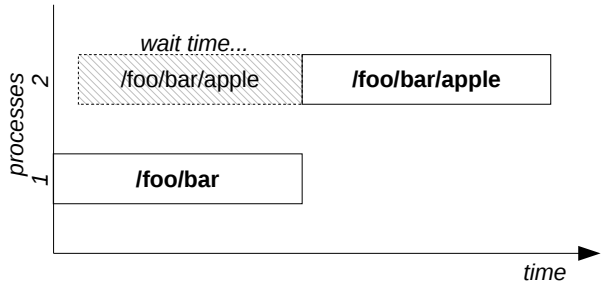

Figure 6: hierarchical resource example

Sequence resources

Sequence resources combines two functions: process synchronization and retrieval of a sequential unique identifier at the same time. Many times sequential unique identifiers are generated using a relational database, but sometimes the full power of a relational database is not necessary or hurts. *Sequence resources* inherit a characteristic from *numeric resources*: a limited number of requesters can obtain a different sequence id at the same time. *Sequence resources* can be "transactional": in the event of execution failure, the unique identifier can be assigned to a different requester. The only supported *lock mode* is "exclusive".

Example: the following command lines gets different sequence ids

```
flom -r _s_id[1] -- echo
flom -r _s_id[1] -- echo
flom -r _s_id[1] -- echo
flom -r _s_id[1] -- echo
...
1
2
3
4
…
```

Figure 7 shows the behavior of a *sequence resource* that allows the concurrent execution of 4 distinct processes: the fifth requester has to wait for the completion of the second one because no free slots are available.

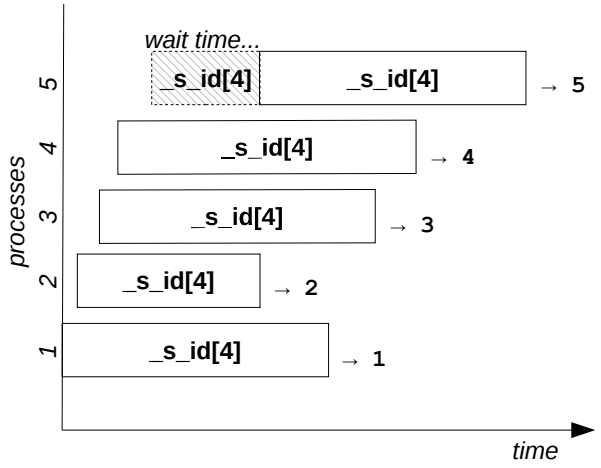

Figure 7: multiple sequence resource example

Figure 8 shows the behavior of a *transactional sequence resource* that allows the concurrent execution of 4 distinct processes: the second requester crashes and its unique id (value=2) is recycled and passed to the fifth requester.

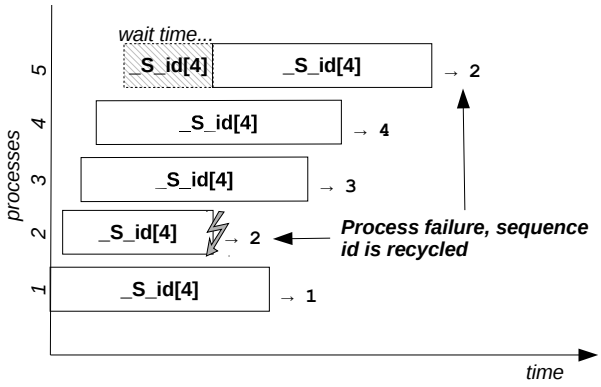

Figure 8: transactional sequence resource example

Timestamp resources

Timestamp resources combines two functions: process synchronization and retrieval of a unique timestamp at the same time; they are similar to *sequence resources* but they provide a unique timestamp instead of a unique identifier and they can't be transactional.

Timestamp resources inherit a characteristic from *numeric resources*: a limited number of requesters can obtain a different unique timestamp at the same time. The only supported *lock mode* is "exclusive".

Example: the following command lines obtains different timestamps

```
flom -r "_t_bar.%x.%X[3]"
 -- echo
flom -r "_t_bar.%x.%X[3]"
 -- echo
flom -r "_t_bar.%x.%X[3]"
 -- echo
…
bar.07/05/16.10:39:01
bar.07/05/16.10:39:02
bar.07/05/16.10:39:03
…
```

Figure 9 shows the behavior of a *timestamp resource* that allows the concurrent execution of 3 distinct processes.

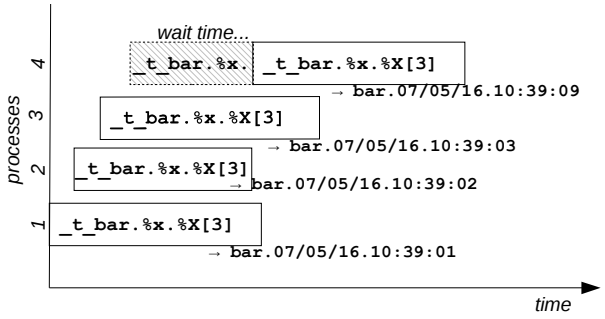

Figure 9: timestamp resource example

Configuration

FLoM implements a multiple layer configuration: an option configured in a higher layer overrides the values configured in the lower layers.
Figure 10a shows the configuration layers available for the `flom` utility (both client and daemon).

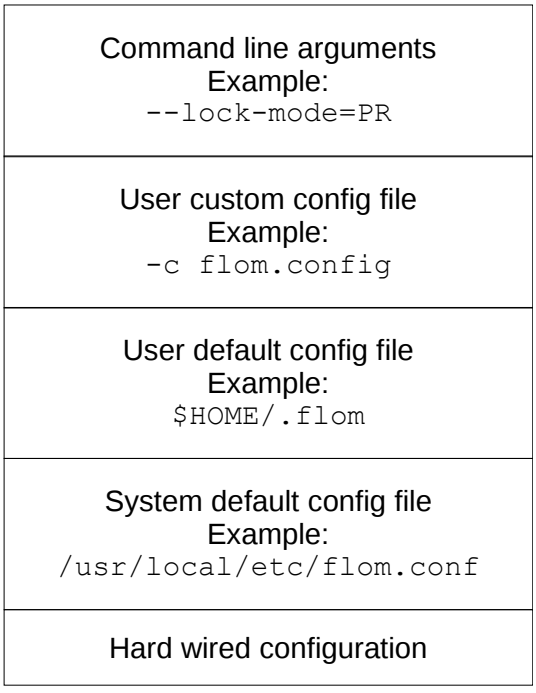

Figure 10a: values defined in the system default config file override hard wired default values; … ; values passed as command line options override all the previous ones.

Figure 10b shows the configuration layers available for `libflom` library:

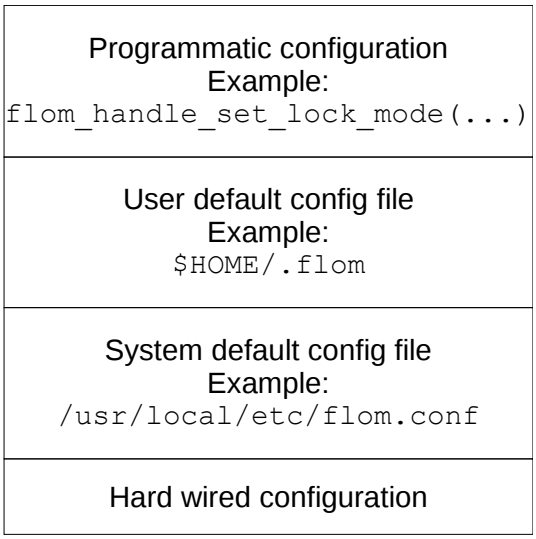

Figure 10b: values defined in the system default config file override hard wired default values; values defined in the user default config file override values defined in the system default config; … ; values set using the API override all the previous ones.

Configuration example

The following examples show three different ways to configure the same option.
The name of the resource that must be used can be specified in a configuration file using the "Name" key inside the "Resource" section:

```
...[3]
[Resource]
Name=MyResource123
...
```

Using the `flom` utility, the resource name can be configured using the command line arguments:

```
flom --resource-name=MyResource123
-- command_that_must_be_executed
```

Using the `libflom` library, the resource name can be configured using the setter method:

```
flom_handle_set_resource_name(my_h
andle, "MyResource123");
```

Technology overview

A detailed description of FLoM internals is outside the scope of this introductory paper, but some information are useful to deploy the technology in the best way.

3 The dots are not part of the file syntax: in this example they must be intended like "rows before" and "rows after" the interesting ones

Daemon internal architecture

The `flom` *daemon* is a multi-threaded process with one listener thread and a distinct thread for every resource[4].

The usage of a dedicated thread for each resource could be considered a waste of computing resources, but it carries three main benefits:

- `flom` *daemon* uses a very low number of explicit synchronization system calls: `flom` *client* requests are intrinsically serialized by socket functions
- `flom` *daemon* latency is minimized: when a request for a resource arrives, there are only two possible delay conditions. The first one is the lack of an available CPU thread, the second one is the execution of a previously arrived request for the same resource
- `flom` *daemon* automatically exploits vertical scalability: the higher the number of CPU threads, the higher the number of concurrent synchronization operations.

All the communications between the `flom` *client* and the `flom` *daemon* use a custom, non blocking, XML based, message passing protocol.

Persistency and consistency

The `flom` *daemon* does not manage persistent data: everything is kept in memory (RAM).

The lack of long term persistence could be considered a limit, but under the assumption that lock state is ephemeral[5], a memory only design carries at least two benefits:

- in the event of a crash, `flom` *daemon* can be restarted without any special attention[6]
- the synchronization process is not affected by disk latency

Every distributed system has to face the constraints described by the CAP theorem[6], and FLoM is no exception. One of the most serious consistency issues related to the FLoM technology is related to *network partition*: under some circumstances, two different, but equivalent `flom` *daemons* could be activated inside a network of distributed systems.

Figure 11a shows a possible distributed initial state with one server (*daemon*) and three clients nodes.

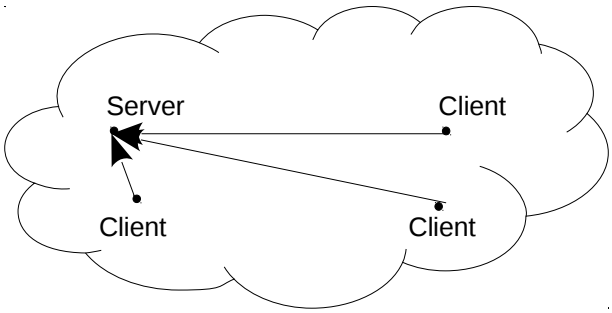

Figure 11a: example of a distributed system before a network partitioning event

Figure 11b shows the result of a network partition if the daemon implicit auto start feature is disabled: the two clients in the right side of the network disconnects from the server and errors are raised.

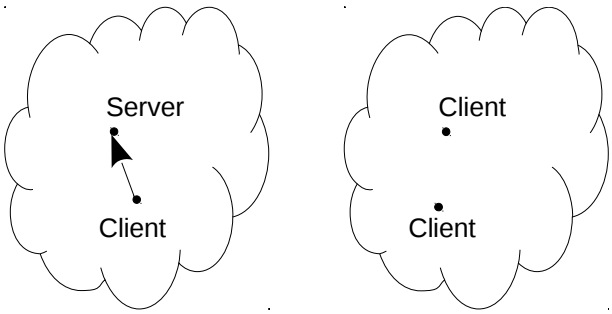

Figure 11b: the clients in the right side of the network cannot connect to the server and raise an error

Figure 11c shows the result of a network partition if daemon auto start and auto discovery[7] features are enabled: in the right side of the network a second daemon can be started. This event can potentially be an inconsistency issue because the two network partitions evolve without coordination: sometimes this is the desired behavior, sometimes this is a dangerous behavior.

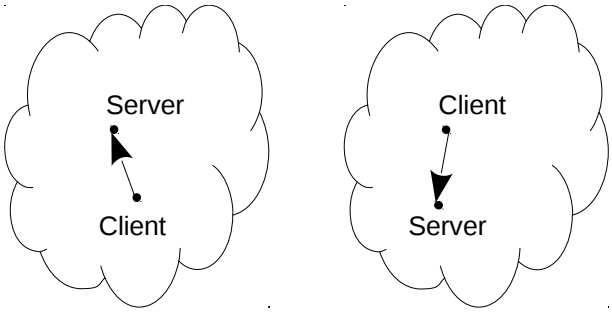

Figure 11c: the two network partitions independently evolve

Security

The FLoM project chose to implement security using the standard features of SSL/TLS. Data transmission is protected using cryptography, peer to peer handshake is authorized using mutual authen-

4 *Hierarchical resources* need just one thread for each distinct root

5 Long lasting locks are typically not compatible with scalable and high performance systems

6 Clients can recognize a server crash checking the return code: in the event of a server crash after the *lock* and before the *unlock* operation, an error code will be returned to the client by the *unlock* operation.

7 The auto discovery feature is implemented using IP multicast and allows the clients to automatically discover the IP address of the server.

tication. TLS mutual authentication introduces a management burden, but it removes all the issues related to *secret* management.

Networking

The FLoM project supports IPv6 that can be used, in conjunction with SSL/TLS security, to deploy *in the wild*: Internet connected devices can communicate in a secure way without auxiliary security like a protected VPN.
The usage of UDP/IP multicast, both IPv4 and IPv6, provides to FLoM a useful auto discovery feature that can be used for *cloud* deployment: `flom` *daemon* can be activated on systems with dynamic IP addressing.

Testing

The FLoM project distributes a complete set of tests along with the source code. The automatic test set is based on the GNU autotool suite: it's used to perform full regression testing after every release and to certify the supported platforms.
FLoM is a software developed for the Linux operating system: it's tested and supported for both Ubuntu and CentOS.

Comparison with similar tools

The author is not aware of a competing tool. Tools designed for the shell environment like `lockrun` [7], `halockrun` [8] and `getlock` [9] use an auxiliary file to manage the locking part of the synchronization process: the design choice limits the usage to the local system or requires the usage of a shared NFS volume to host a lock file shared among many systems. All the tools provides only a command line interface: an API is not available.

Conclusions

This brief paper describes the basic characteristics of an open source and free software project that provides synchronization tools for:

- shell commands and shell scripts executed from the command line or the *cron* Linux standard service
- custom programs developed by the user using one of the following programming languages: C, C++, Java, PHP and Python

Leveraging the *resource* concept, FLoM can be used to implement both trivial and complex synchronization use cases among shell commands (scripts) and custom programs.
The performance, scalability and security features provided by FLoM enable a wide range of different deployments from a single system to a complex network of geographically interconnected systems.

References

[1] FLoM on GitHub:
https://github.com/tiian/flom
FLoM on SourceForge:
https://sourceforge.net/projects/flom/
FLoM documentation:
https://sourceforge.net/p/flom/wiki/Home/
[2] Glossary of *The Reactive manifesto:*
http://www.reactivemanifesto.org/glossary#Scalability
[3] SWIG:
http://www.swig.org/
[4] DEC OpenVMS DLM:
https://en.wikipedia.org/wiki/Distributed_lock_manager
[5] FLoM resources:
https://sourceforge.net/p/flom/wiki/Resource/
[6] Seth Gilbert, Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, Volume 33 Issue 2, June 2002, 51-59
[7] lockrun home page:
http://www.unixwiz.net/tools/lockrun.html
[8] halockrun home page:
http://www.fatalmind.com/software/hatools/halockrun.man.html
[9] getlock home page:
https://sites.google.com/site/columscode/home/getlock

Acronyms and abbreviations

API: Application Programming Interface
JNI: Java Native Interface
LAN: Local Area Network
MAN: Metropolitan Area Network
NFS: Network File System
VPN: Virtual Private Network
WAN: Wide Area Network