Socketeer – a proxy with twist

(POC version alpha 2 – proof of concept)

POC a2, document version 2

author: Alen Milincevic

Tablica sadržaja

Why the name?

"Musketeer" is the name originaly used for an early type of infantry soldier with a musket.
A musket is a type of the firegun.

SOCKS is the name for the pervasive protocol operating on the session layer of OSI model.
It is commonly used for connecting through firewalls, or to be a part of a firewall.

SOCKET is used in the sense of a network socket, which is used for interprocess flow in a computer network.

"Socketeer" is a word which combines SOCKS/SOCKET with musketeer.

Why it is concieved?

The author had a problem with MariaDB Galera connecting through different subnetworks.

MariaDB Galera uses some particular reserving of the port "4444".
While there are several solution to some exotic configurations, none is elaborate enough.

Socketeer aims to make it straitforward.
While analysing the problems, there are many more possible computer related problems, which Socketeer could also solve.

One JAR file and a configuration on both endpoints connected through a firewall robust tunnel would do the trick, if the configuration is properly set.

Purpose

For any host behind a firewall it is trick to use server ports. SOCKS protocol as such allows usage, but is limited to one server connection client.

After long thinking of the problem, the obvious solution would be to make two proxy interfaces, which apper like being server proxy working one-way. A kind of making a full two-way VPN, but on the application level. No TUN/TAP kernel level messing around and no root access required.

Java is a good solution to make a portable, self contained VPN. It has many features already built in and also included built-in SOCKS chaining ability.

Arhitectural decision

The original problem was this:

1 cluster of 2 (later 3) virtual machines, which could only accessed via ssh tunnel.
SSH tunnel supports local port forwarding, remote port forwarding and dynamic forwarding (SOCKS proxy) from a SSH client to this machine.
When the MariaDB instances with Galera enabled run inside the cluster, everything works.
An obvious solution could be to set a proper VPN with outside world, in order to have more nodes to test.
For some reason, that was not an option in this case. It would require to deep intervention into the running system.

There was a need to test behaviour of "brain split" situation with more nodes.
As there were only 2, later 3 virtual nodes, to bring more nodes into the test requires some kind of full VPN-like functionality.

At the end, another solution was tried out (a virtual machine in an outside computer, with identical configuration, but with a cluster of virutal computers run inside a virtualizing solution).

As the aftermath of this problem, the idea of Socketeer was born, in order to not have such problems again in the future.
This made the author of Sockseteer thinking hard for a portable and simple solution, one which would not require setting no TAP device drivers for VPN and would intervene in the running system as little as possible.
The machines had Java installed.
Also, there was the possibility to probably socksify MariaDB. And probably some other applications.

Thinkering about a "simple" solution

After having given much thought about a kind of two SOCKS servers connected "back to back" and having many reserach in existing solutions, none was found in this direction.

However, the author of Socketeer had already expirience with some messaging servers.
Messaging servers work by receiving posted messages to queues and topics and receivers (JMS protocol and STOMP to some degere) and then distributing them. This simple idea is much more powerfull if used in connection with the SOCKS proxy idea.
Messaging means that messages can go in any direction over a single connection (typically TCP/IP is used), and that makes a kind of "connection-multiplexing".

Having that idea now seems obvious (althorough needed much imagination work to concieve in this form).

After searching for simple solutions and doing much "trial and error", a very simple Java STOMP sever class was chosen. STOMP in itself is very powerfull, but only the basic functionality would sufice:

- send message
- receive message

Using it wisely and on two sides of a firewall (which could allow only one outgoing port), anything can be acted upon a message, by doing this:

- receive message
- parse message and do something with it
- send the result to the original sender

In example, a SOCKS end could send a message, i.e. "open remote socket", which is parsed, acted upon, and a message returned. Also, when some data is available, this is also sent as a message. Closing of sockets, ping-pong and remote resolving, all that can be done as signalling, over one message topic.
In fact, anything could be encapsulated inside, even more than two endpoints can be connected, if everyone has own unique id. Any can be programmed to act only on messages relevant to itself and to send new messages as answer, which are the result of the original message.

Having many years of expirience with network topologies, different protocols, different programming languages and software design, this was chosen over more complex alternatives.

Old problems possibly also solved

The author had problems in the past. Where there are problems, a solution must be seeked. This is solution oriented thinking. When there is no software which does not (adequately) solve the required task, one must make it by himself.

While working for a software company which had a very strange internet policy and internet was required for daily work, some alternatives were searched.
Since the official company policy stated, that "any kind of tunneling" was prohibited, another way to acces the vital content needed for work was seeked. Sure, one could download required material at home via internet. Sometimes this was not possible due to time constrains. So, the idea of "e-mailing list" was born. While technically this would have been tunneling http over e-mail, it could be easly also be justified as "e-mail list". There was a home computer, which could have been left working on a cable flat-rate connection. Outside of any firewall. At this time the author did not have enough expirience and skills to come with a specific concrete qualitative solution. There were some failed experiments with a Pascal mail bridge with GUI (finally finished last version, albeit the author than left the company and the problem vanished).

Another problem was for a game, which could not be played, since it used an XML protocol on a port other than 80. In corporate environments it could not be played, if only port 80 was allowed on firewall. The protocol used could be intermingeled with HTML, if a kind of "content based filtering" was advised. On the same port. The solution was worked on and finished as being called "sf-rmr", another program. It had it's shortcomings with bidirectionality and binary support.
The game was finally abadoned in it's concept as such for other reasons (mainly it failed to attract enough players).

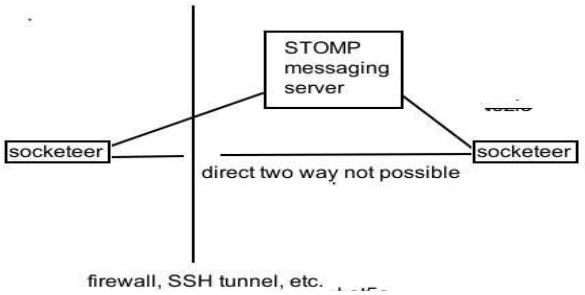
Third problem was one with clustering behind a firewall and the possibility to make a "make-shift" test without intervening to much in the system. If intervening, than possibly with minimum impact on any other proces running on the cluster.

There is actually one more problem regarding a quiz running on the IRC. It run on a faculty owned server, which had IRC enabled. SOCKS could be supported. After some time the political decidion came, to not allow IRC, only HTTP. A very restrictive firewall in fact cut the idea.
At that time (this would have been problem zero, a long time ago), first thougths came up, how to circumvent this. At that time, which is very long ago, not enough knowledge nor existing solution were present.

Socketeer is the third attempt, which tackles all of the problems above and seems to have some more additional potential. It strives to be an universal solution for even more anticipated future problems.
It is significant, that there also were three attempts into socketeer (one was a general "proof-of-concept" which failed in multithreading, the second was a second rewritten and cleaned up same concept with to much recursivity and stack usage, and the third is built on the experiences without the shortcommings of the second concept, especially new concept of a "pool", which eases debugging much more).

Simplified functionality

A TCP/IP connection is a twoway connection. It can send and receive, all on one connection.
It is specified by a host and port pair. On one host, there can not be more than one server on the same port. However, a physical device can have more than one IP address. Therefore, on one physical device it is possible to have multiple host:port combinations, when the host part is different. Furthermore, with upcoming of virtual machines, virtual interfaces and virtual hardware in general, this can get more complicated.

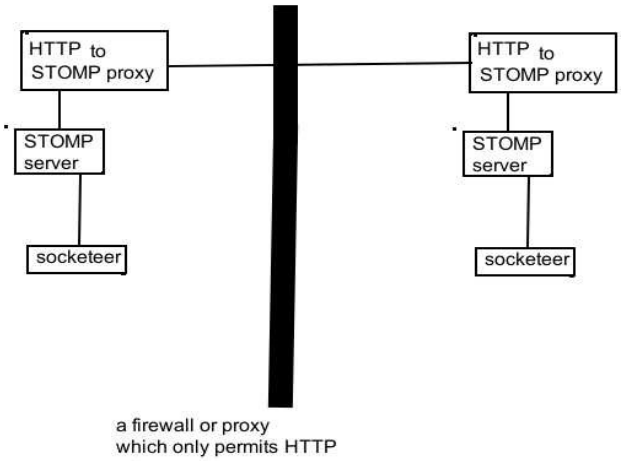


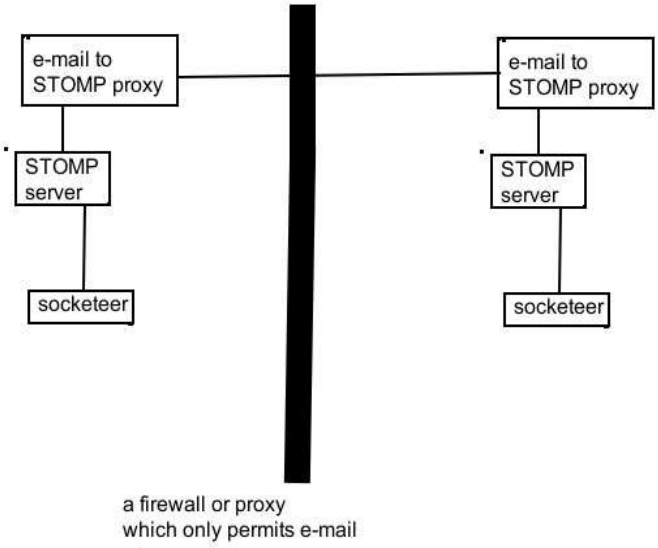
Firewall is a device, which controls access. Usually this means, that server sockets are not allowed. Socketeer is a server which binds to a host:port, and using SOCKS specification (primary) or some other exotic proxy-ing techniques pretends to the outside world to be a "back-to-back" server. Each side can see on other side the hosts, which are allowed through configuration. STOMP messaging server acts as a mediator and is placed usually inside firewalled environment. STOMP server acts as a bridge, a kind of VPN (virtual private network), but only using TCP sockets. On the side where STOMP server is located, it must have the permission to use the server port. Usually, on Unix-like operating systems, ports under the port number 1024 can only be opened by the root user.

Advanced use cases

It would be possible to use also firewalls, which only permit e-mail or HTTP(s) requests.
Most common configuration is to permit only HTTP(s).
Using of e-mail to go through can be regarded as a kind of "mail-list" functionality.

I.e. :





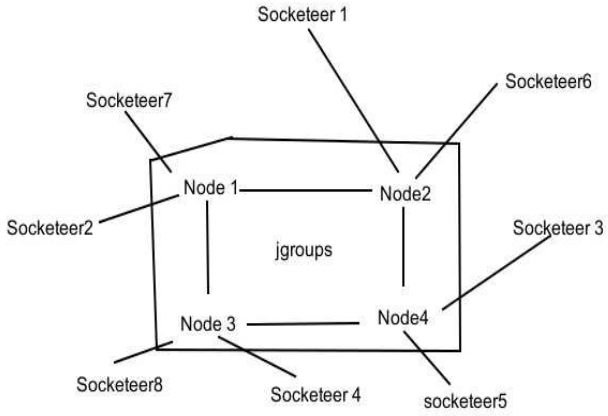
"http to STOMP proxy" and "e-mail to STOMP proxy" do **not exist yet**. This is only a **conceptual idea**. There are some very usefull tools in Java (one being also "Apache Camel"), which already have much of the functionality of bridging connections. In fact, Apache Camel was considered as the first choice, but using a small in-built STOMP server was chosen due to more simplistic approach. Furthermore, Apache Camel (or any other similar STOMP bridge) can easly be integrated on top of STOMP in a layered approach.

Reaching out in integration

STOMP being a Simple (or Streaming) Text Oriented Message Protocol makes it very versatile and powerfull. There are some tools, which can extend the reach of a proxy "back-to-back" solution in many ways.

Some of them being:

- socat - SOcket CAT - multipurpose relay - Linux / UNIX TCP Port Forwarder – a powerfull tool which, among other things, can map a socket to a physical or virtual serial port. In modern Linux-like operating systems it is trivial to open new virtual serial ports. Windows operating system requires additional drivers to accomplish this task.
- jgroups – a reliable multicast system, written in Java, which also has STOMP serving support. This system has a very versaitile routing system, so complex solutions could be provided using jgroups in conjunction with Socketeer. A complex example of connecting multiple socketeers into a jgroups channel:



- com0com – a product kernel-mode virtual serial port driver specially for Windows. This product works by creating pairs of serial ports, where two ports are connected one to another. This enables applications, which work over serial line, to have mutual input and output one to another. On Linux-like operating systems this task is usually much more trivial so this is not required on them.
- Apache camel - a rule-based routing and mediation engine. Very powerfull library. Needs also a concrete implementation, if it is to be used with Socketeer in some way.

Building blocks of the Socketeer solution

In normal client/server internet communication, there is a server side and one or more client sides. However, in the socketeer idea, there is the concept of "**sourcesinks**", since every endpoint can be either source or the sink (of a SOCKS relay or otherwise).

The concept is to have at least one sink (outgoing connection) which is connected to a STOMP server. Source is the TCP/IP SOCKS entry point, which is spawned for every newly connected client.

The general idea is to have to possibility to form clusters, which are comprised of one source and sink. Technically it is possible to have a cluster of joined more than one source and sink, but this is considered as a possible extension in the future.

Every sink and source in a cluster must have an unique name.

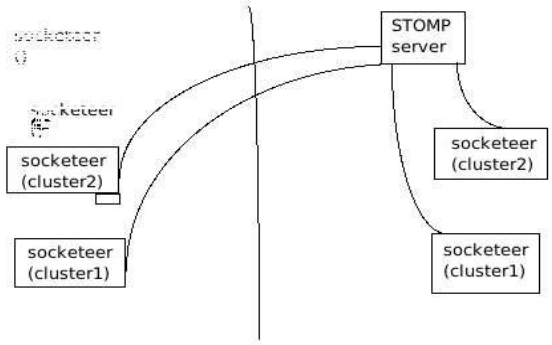
The cluster name is common for all the sourcesinks in the same cluster.

Every sourcesink has one or more "resource"-s (typically socket endpoints of a connection).

In comparison to TCP/IP protocol, where the building blocks are nets,subnets,hostname and port combination, here it is the combinantion of cluster,sourcesink node and resource which identify a resource.

Messages

Any communication between sourcesink blocks in a socketeer cluster is performed by using messages. Messages are passed via a TCP/IP connection to a messaging server. As messaging is a very powerfull concept, many topologies are possible. i.e. :



Messaging servers usually work with the concept of a sender and a receiver. In order to be able to identify an unique sender and receiver, every one of them needs own unique address. In socketeer, the usage of the very robust TCP/IP protocol is utilised, and the encapsulation is made within STOMP protocol. In TCP/IP, the unique address is host and port. With IPv4, there are also subnets, but with IPv6 it is also possible to have much more address ranges.

Socketeer protocol unique address is composed of cluster ID,sourcesink node ID and an unique resource ID.
Every message is sent from a Socketeer sourcesink towards another, or to the whole cluster (broadcast). The messages contain timestamps, originating sourcesink, destination sourcesink, sequence id and the payload.

While it would be feasible to extend them with a control code (to veryify integrity) and/or encryption, this are not neccessarly elementary required functions.

Unique addresses also mean, that many clusters could be used on a single server, each cluster using only it's own processing. They would not intermingle, since the addressing sheme is unique. In programming concepts, clusters are some kind of namespaces.

Messages are built as a textual datagram, which is partially URLencoded.
While other traffic could also pass on the same "topic", it is advisable to reserve only one for each. Socketeer will ignore traffic, which is not a known textual datagram.

Core socketeer protocol message types

- open TCP socket
- send data over TCP socket
- close TCP socket
- ping/pong
- resolve
- ... more planned ...

Property files and profiles

As Java has a wellthought concept of property files, it is logically to use this mechanism for configuring. Socketeer message has also a part based on property mechanism.
The well thought mechanism of properties in conjunction with java generic stream functionality allows serialization, transmission over various media and easy handling as a generic key and property pair. Java has also a wellthough URL encoding/URL decoding mechanism based on the open specification available on the Internet. Java properties can be URL encoded in serialisation, which is convinient for some old TCP transmission methods, which "chocke" on special characters. This is the main reason that URL encoding was invented.

Many Java based frameworks use properties as their underlying mechanism. The concept of profiles allows reuasage of complete blocks of properties, which are referenced by the name of the profile. Properties can be easy read in and read out from a file, socket, command line (i.e. URL encoded), a unnamed or named pipe (on Linux, if URL encoded) and from any generic Java InputStream and to any generic Java OutputStream.

Socketeer defines profile names for sourcesinks, for blacklists, whitelist, maplists, authorisation.

The names of profiles may be URL encoded, but must not contain the character '.' in their name. This character is used as a delimiter. However, they can contain this character as URL encoded value.

The basic profile is the profile which defines a sourcesink, more specifically a source or a sink type. The type of the profile (if the sourcesink is used as a source or as a sink) is set in the Java properties itself. Also, there are links from the sourcesink profiles to other profiles. The reason for linking is the reusage of blocks of properties (i.e. shared lists btween sourcesinks).

The way Socketeer is thought of is to be universal and flexible, it can have any number of Sources or Sinks running on one physical (or virtual) TCP host or be distributed in any manner.

Socket chaining and proxy chaining

Socket chaining is a concept of connecting multiple proxies into a chain, whereas every one can take specific role in filtering. As Java has native support for SOCKS4(a)/SOCKS5, this proxying is nateivly supported. This can be configured via java system properties. See more relevant documentation on this issue. Socketeer can be integrated into such a chain, as it is written in Java. It is possible to use a backend or frontend proxy (i.e. jsocks) to accomplish more usecases, i.e. better UDP relaying through jsocks. It is a good programming phylosophy not to "reinvent the wheel", but instead combine existing solutions into a new and more flexible better one.
Proxying support for SOCKS4(a)/SOCKS5 in Java is however of varing degree.
Depending on the version, early ones support only SOCKS4, and newer ones dropped official support for SOCKS4 and support only SOCKS5.

Proxy chaining in general is the connection from multiple proxies in a chain. A good use of this is anonymity. It is also used for evil purposes, however, anything can be used for good or for evil. Using of proxy chaining enables very good and even fail-over network topologies.
Socketeer can also be chained to another instance of itself.
It can also be chained to i.e. sf-rmr, in order to get additional functionality.

Althorough SOCKS specification postulates that GSS-API "must" be implemented by SOCKS proxies, it is often not the case, and not every proxy support it. Making the support for it mandatory is very bad, since this is not so trivial to implement programatically as is no authentification or just username and password authentification. Through chaining, GSS-API could be supported by some upperlayer SOCKS server, which chains over some secure channel to Socketeer and authorises with some complex username/password, that are somehow protected from eavsdropping beyond reasonable doubt to be to easy broken. This is the phylisophy "just good enough", not to make matters to complex. As GSS-API authorisation is not used as much as the authors of specification hopen (most researched SOCKS proxies use no authentification, just a few use username and password authentification). There are also access lists in socketeer, so the access can be regulated in many ways already as it is. Having a firewall, which does not even allow certain IP's, or allows only certain IP's is an additional option.
This solution is stackable.

Concept of socketeer channels

Socketeer channels are ment to be a generic way to connect sourcesink's.

An example would be:

stomp:localhost:61626/sisotopic/?username=user&password=pass&codekey=blabla

This channel universal resource locator would use *STOM*P protocol (possibly gozirra STOMP server), post messages and listen to *sisotopic*, using the username (*"user"*) and password (*"pass"*). The username and password are the ones used on the STOMP server as access data.
The communication is also encrypted (using jasypt) with a simple simetric key. This encription is not a strong one, however better than no encryption. For now, only stomp is supported. If possible, it is planed to implement other protocols, the first most obvious would be HTTP/S. Maybe also some serial port connectivity support (virtual com ports exist for a few platforms and also port redirectors exist for a few platforms).

Sample config file

This is a sample *socketeer.properties* :

socketeer.general.messagingserverenabled=true

socketeer.general.messagingserverport=61626

#TODO

#socketeer.general.sosisize=

#socketeer.general.relaysize=

TODO

#socketeer.general.messageencryptionkey=

socketeer.sourcesink.0.clustername=c

socketeer.sourcesink.0.nodename=si

socketeer.sourcesink.0.channel=stomp:localhost:61626/sisotopic/?
username=user&password=pass&codekey=blabla

socketeer.sourcesink.0.isSpawningAndResolvingSink=true

socketeer.sourcesink.1.clustername=c

socketeer.sourcesink.1.nodename=so

socketeer.sourcesink.1.channel=stomp:localhost:61626/sisotopic/?
username=user&password=pass&codekey=blabla

socketeer.sourcesink.1.serverport=1234

socketeer.sourcesink.1.isSpawningAndResolvingSink=false

socketeer.sourcesink.1.peernode=si

socketeer.sourcesink.1.fixedHost=localhost

socketeer.sourcesink.1.fixedPort=80

socketeer.sourcesink.1.fixedType=0

access list

#socketeer.sourcesink.1.accesslist=0

#socketeer.accesslist.0.host=129.0.0.1

blacklist

#socketeer.sourcesink.1.blacklist=0

#socketeer.blacklist.0.cluster=c

#socketeer.blacklist.0.node=si

#socketeer.blacklist.0.host=localhost

#socketeer.blacklist.0.port=80

whitelist

#socketeer.sourcesink.1.whitelist=0

#socketeer.whitelist.0.cluster=c

#socketeer.whitelist.0.node=si

#socketeer.whitelist.0.host=localhost

#socketeer.whitelist.0.port=80

```
# maplist
#socketeer.sourcesink.1.maplist=0
#socketeer.maplist.0.orghost=localhost
#socketeer.maplist.0.orgport=80
#socketeer.maplist.0.maphost=localhost
#socketeer.maplist.0.mapport=90

# proxy init
socketeer.sourcesink.1.socks4enabled=true
socketeer.sourcesink.1.socks5enabled=true
socketeer.sourcesink.1.httpenabled=true
```

Usage

Java -jar socketeer.jar [config filename]
Java -jar socketeer.jar [*TBD*] [properties source]

TBD. To comment better on configuration options.

Intended usage

As with every software, the usual disclaimer applies. This software was created for the purpose of good. It is authors strong belief, that when a being shares good things, more good things come.
It is strongly discouraged for a usage which would harm anyone or anything.
The author will not and cannot be responsible for any harm with this specific software.
Be advised that the usage is without a warantee for any purpose.
The author can find most use for it in development environments (as he usually needs simulated functionality).
Made with good will, ment to be used for good.

Other planned features

- encachanged maplist (possibly also good for IPv4/IPv6 remapping funtionality)
- better HTTP CONNECT and HTTP proxing possibility
- full SOCKS5 support (*correct binding responses, UDP*).
  Currently only SOCKS4/SOCKS5-like interface, with a CONNECT like interface and HTTP proxy basic methods. All UDP packets are always silently dropped (as allowed by RFC).
- "content-based" connecting, based on the idea and experiences of **sf-rmr**

- other protocol support besides STOMP (probably HTTP and e-mail, possibly also raw serial port and named pipes support)
- Simplifing code, right now to complex
- Message encription techniques, better usage authorisation and more robust protocol
- more documentation, extending features, real world applicable examples
- built-in support for IRC, XMPP and possibly other messaging protocols. In Java, good quality libraries already exist, only the implementation using integration would need more debugging time.
- possible Apache Camel integration
- complete rewrite, reducing redundant code, making even more flexible

References

- tunneling protocol - http://en.wikipedia.org/wiki/Tunneling_protocol
- proxy server - http://en.wikipedia.org/wiki/Proxy_server
- SOCKS 4/5 - http://en.wikipedia.org/wiki/SOCKS
- TCP, UDP and other protocols - http://en.wikipedia.org/wiki/Internet_protocol_suite
- tsocks, proxychains, dante, sockscap, widecap... (and other programs known as "proxyfiers") - http://en.wikipedia.org/wiki/Comparison_of_proxifiers
- a good article on proxy chaining: http://hackershandbook.org/tutorials/proxychaining
- virtual private network - http://en.wikipedia.org/wiki/Virtual_private_network
- STOMP - http://en.wikipedia.org/wiki/Streaming_Text_Oriented_Messaging_Protocol
- sf-rmr - https://code.google.com/p/sf-rmr/ , http://www.slideshare.net/AlenMilincevic/sf-rmr-servicing-forwarding-remote-multiplexing-relay
- jsocks - http://jsocks.sourceforge.net/ , http://code.google.com/p/jsocks-mirror/ , https://github.com/ravn/jsocks
- jasypt - http://www.jasypt.org/
- socat - http://www.dest-unreach.org/socat/
- jgroups - http://www.jgroups.org/
- com0com - http://com0com.sourceforge.net/
- Apache Camel - http://en.wikipedia.org/wiki/Apache_Camel
- Gozirra – https://github.com/pedroteixeira/gozirra, http://web.archive.org/web/20130911060110/http://www.germane-software.com/software/Java/Gozirra/