Implementing a RFC 3261 SIP Registrar
A quick-starter's guide

by Michael Maretzke

26th July 2007

Introduction

The RFC 3261[1,2] defines the SIP protocol's syntax, its semantics and basic usage patterns. In chapter 10, the process of "Registration" and its meaning for the protocol. The RFC introduces an entity named Registrar. "*What does a SIP Registrar do? How can it be implemented? Is an open-source Registrar available?*"
These questions are addressed in this paper – which describes an actual implementation of a RFC 3261 compliant SIP Registrar being implemented in Java Technologies utilizing the SIP Servlet[3] programming model.

What does a SIP Registrar do?

The SIP protocol offers a discovery capability. This discovery process is initiated if a user wants to establish a session with another user utilizing the users' SIP URI. Proxy servers or redirect servers involved in the setup of the session reach out to Location Services to resolve abstract SIP URI's (Address Of Record, AOR) like e.g. "sip:eduardo@thesipdomain.de" into real existing termination endpoints e.g. "sip:eduardo@192.168.178.200:7060". So, the concept of the Location Service is quite comparable to a Domain Name Server (DNS) where domain names are resolved into IP addresses.
The Location Service's information store is populated and updated through the domain's SIP Registrar which extracts relevant information from the user's SIP REGISTER messages.

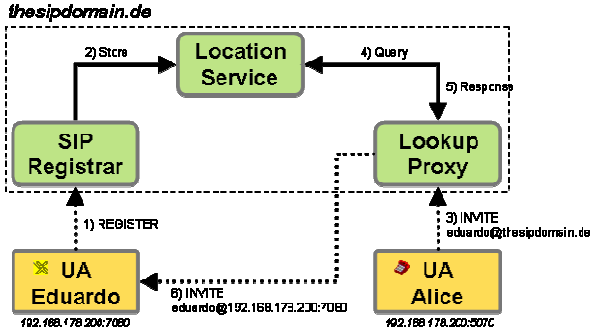

Figure 1 Flow of information in a SIP Registrar

The discovery concept of the SIP Registrar introduces mobility and flexibility. The user Eduardo could register his current location through the SIP Registrar at the Location Service of "thesipdomain.de" and remains reachable via his SIP address "sip:eduardo@thesipdomain.de" independent of his current geographical location. In addition, he may assign multiple termination points – contact addresses – where he is reachable via the before mentioned abstract SIP address.

Elements of a SIP Registrar

Figure 1 shows three main functions: the SIP Registrar, the Location Service and the Lookup Proxy.
The SIP Registrar deals with incoming SIP REGISTER messages sent by user agents. These SIP messages contain information about the binding of abstract SIP addresses to real existing contact points.

```
REGISTER sip:thesipdomain.de SIP/2.0
Via: SIP/2.0/UDP 10.8.1.109:5100;branch=z9hG4bK-1-0
From: Alice <sip:alice@thesipdomain.de>;tag=1
To: Alice <sip:alice@thesipdomain.de>
Call-ID: 1-5580@10.8.1.109
CSeq: 1 REGISTER
Contact: <sip:alice1@10.8.1.109:5100>
Contact: <sip:alice3@10.8.1.109:5111>;expires=3600
Contact: <sip:alice2@10.8.1.109:51001>;expires=1800
```

Figure 2 SIP REGISTER message

The SIP REGISTER message above maps the abstract SIP address, the address of record (AOR), "sip:alice@thesipdomain.de" served by the SIP Registrar of "thesipdomain.de" to the concrete user agent endpoints listed at the "Contact:" headers:

- `sip:alice1@10.8.1.109:5100`
- `sip:alice3@10.8.1.109:5111`
- `sip:alice2@10.8.1.109:51001`

The SIP Registrar extracts all valuable information from the incoming SIP REGISTER message and hands it over to the Location Service.
The Location Service stores or updates the information received by the SIP Registrar and offers query capabilities to the Lookup Proxy.
The Lookup Proxy works on received SIP INVITE messages targeted at the served domain "thesipdomain.de". The INVITE messages are targeted at the abstract SIP addresses and need to

be resolved into the current reachable user agent of the user. So, the Lookup Proxy queries the Location Service and proxies the received SIP INVITE message to the correct user agent termination point.

RFC language translated

The chapter 10 of RFC 3261 describes the functions provided by a SIP Registrar in great details. To give the reader an overview of the functions a walk-through of the most relevant part of the RFC 3261 is discussed here: the step-by-step description. The description points also to the relevant sections in the source files of the example implementation. In the RFC language, a binding maps the Address Of Record (AOR; the abstract SIP address) towards one or multiple contact addresses. Furthermore, the binding stores some further information like e.g. the expiration time of each contact address, the relative prioritization, the Call-ID, the call sequence number and some more. The implementation's pendant of a binding can be found in `Registration.java`.

```
When receiving a REGISTER request, a registrar follows these
steps:

1. The registrar inspects the Request-URI to determine
whether it has access to bindings for the domain identified
in the Request-URI. If not, and if the server also acts as a
proxy server, the server SHOULD forward the request to the
addressed domain, following the general behavior for
proxying messages described in Section 16.
```

The Request-URI of the REGISTER message is contained in the first line of the SIP REGISTER request – "sip:example.de" in the example request shown in Figure 2 below. The URI is served only if it matches the domain of the SIP Registrar – otherwise it is proxied to the target domain.

`Registrar.java: 196ff`

```
2. To guarantee that the registrar supports any necessary
extensions, the registrar MUST process the Require header
field values as described for UASs in Section 8.2.2.
```

The example implementation described here does not support any SIP extensions. So, if the `Require` header is present in the request the SIP Registrar will always return a `420 Bad Extension` response.

`Registrar.java: 214ff`

```
3. A registrar SHOULD authenticate the UAC. Mechanisms for
the authentication of SIP user agents are described in
Section 22.
Registration behavior in no way overrides the generic
authentication framework for SIP.  If no authentication
mechanism is available, the registrar MAY take the From
address as the asserted identity of the originator of the
request.
4. The registrar SHOULD determine if the authenticated user
is authorized to modify registrations for this address-of-
record.
For example, a registrar might consult an authorization
database that maps user names to a list of addresses-of-
record for which that user has authorization to modify
bindings. If the authenticated user is not authorized to
modify bindings, the registrar MUST return a 403 (Forbidden)
and skip the remaining steps.

In architectures that support third-party registration, one
entity may be responsible for updating the registrations
associated with multiple addresses-of-record.
```

The SIP Registrar implementation described here does not include user agent client (UAC) authentication mechanisms.

`Registrar.java: 222ff`

```
5. The registrar extracts the address-of-record from the To
header field of the request. If the address-of-record is not
valid for the domain in the Request-URI, the registrar MUST
send a 404 (Not Found) response and skip the remaining
steps.
```

The validity check in the SIP Registrar implementation is limited to a check if the '@' character is present.

`Registrar.java: 231ff`

```
The URI MUST then be converted to a canonical form.  To do
that, all URI parameters MUST be removed (including the
user-param), and any escaped characters MUST be converted to
their unescaped form.  The result serves as an index into
the list of bindings.
```

The request URI needs to be unescaped. For doing so, the SIP Registrar implementation utilizes the "commons lang" library from the Jakarta Project[4].

`Registrar.java: 240ff`

```
6. The registrar checks whether the request contains the
Contact header field. If not, it skips to the last step. If
the Contact header field is present, the registrar checks if
there is one Contact field value that contains the special
value "*" and an Expires field. If the request has
additional Contact fields or an expiration time other than
zero, the request is invalid, and the server MUST return a
400 (Invalid Request) and skip the remaining steps.
```

The actual syntax of the REGISTER request is checked in the class `Registration`. Herein, the whole request is parsed and evaluated if it's a valid or invalid request.

`Registration.java: 137ff`

```
If not, the registrar checks whether the Call-ID agrees with
the value stored for each binding. If not, it MUST remove
the binding. If it does agree, it MUST remove the binding
only if the CSeq in the request is higher than the value
stored for that binding. Otherwise, the update MUST be
aborted and the request fails.
```

The paragraph above is quite complex to interpret by a reader. It basically describes the conditions to remove a binding as a result of a successful wildcard registration.

`Registrar.java: 254ff`

```
7. The registrar now processes each contact address in the
Contact header field in turn.  For each address, it
determines the expiration interval as follows:

 -  If the field value has an "expires" parameter, that
value MUST be taken as the requested expiration.
 -  If there is no such parameter, but the request has an
Expires header field, that value MUST be taken as the
requested expiration.
 -  If there is neither, a locally-configured default value
MUST be taken as the requested expiration.
```

Each contact address found in the REGISTER message may expire at different times. These

expiration times need to be respected correctly (which most of other implementations don't do). In this implementation the expiration timers are implemented by the Location Service.

`InMemoryLocationService.java: 57`

```
The registrar MAY choose an expiration less than the
requested expiration interval. If and only if the requested
expiration interval is greater than zero AND smaller than
one hour AND less than a registrar-configured minimum, the
registrar MAY reject the registration with a response of 423
(Interval Too Brief). This response MUST contain a Min-
Expires header field that states the minimum expiration
interval the registrar is willing to honor. It then skips
the remaining steps.
…
```

The SIP Registrar implements the behavior of checking the minimum expiration value.

The minimum value can be configured in the `sip.xml` deployment descriptor file packaged with the application.

`Registrar.java: 244ff`

```
<servlet-name>registrar</servlet-name>
     <servlet-class>
          com.maretzke.sip.tools.registrar.Registrar
     </servlet-class>

     <init-param>
          <param-name>Registrar.MinExpireValue</param-name>
          <param-value>10</param-value>
     </init-param>
...
```

Figure 3 Extract of the deployment descriptor sip.xml

```
For each address, the registrar then searches the list of
current bindings using the URI comparison rules.  If the
binding does not exist, it is tentatively added.  If the
binding does exist, the registrar checks the Call-ID value.
If the Call-ID value in the existing binding differs from
the Call-ID value in the request, the binding MUST be
removed if the expiration time is zero and updated
otherwise. If they are the same, the registrar compares the
CSeq value. If the value is higher than that of the existing
binding, it MUST update or remove the binding as above.  If
not, the update MUST be aborted and the request fails.
This algorithm ensures that out-of-order requests from the
same UA are ignored.
…
```

The actual management of the bindings is implemented in the `Registrar` SIP Servlet.

`Registrar.java: 253ff`

```
8. The registrar returns a 200 (OK) response.  The response
MUST contain Contact header field values enumerating all
current bindings. Each Contact value MUST feature an
"expires" parameter indicating its expiration interval
chosen by the registrar. The response SHOULD include a Date
header field.
```

The SIP Registrar implementation constructs a final `200 OK` message if everything went fine. Otherwise, already earlier in the process error messages are constructed and sent out to the initiator of the SIP REGISTER message.

`Registrar.java: 277ff`

Overview of the Implementation

The design of the implementation introduced in this paper is very much aligned with the functions outlined in Figure 1 above. It introduces classes for the SIP Registrar, the Location Service and the Lookup Proxy.

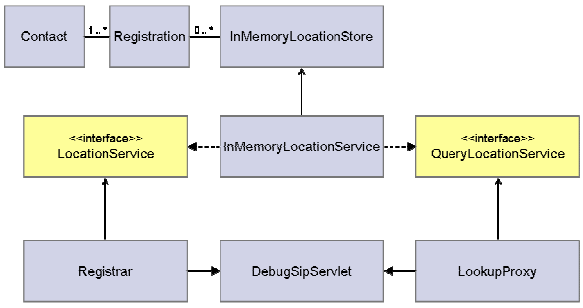

Figure 4 Class overview SIP Registrar

The design aims at a carrier-grade solution. It decouples the Location Service via interfaces from the surrounding elements. The example implementation does not implement a failover safe and high-available solution. The Location Service is implemented as a single-node, in-memory data store. If the instance of the SIP server goes down, all registrations will be lost. In a carrier-grade environment, the Location Service needs to be implemented to allow the distribution of information.
The main focus of the provided implementation is on providing a RFC 3261 functional compliant SIP Registrar implementation.

Why SIP Servlets?

The implementation described in this paper is based on the SIP Servlets programming model. The programming model is very easy to learn due to its similarity to the well-known HTTP Servlet model and is targeted at serving SIP requests – so it's a natural choice. Furthermore, it was intended to use available products – preferable in the Java programming language.
From this perspective, it is quite handy that the BEA WebLogic SIP Server comes for free[5] for development purposes and a lately released open source project named SailFin[6] has been launched.

`DebugSipServlet` is the common base class for the SIP Servlets `Registrar` and `LookupProxy`. The base class offers extended tracing and error response handling methods and extends the base class of the SIP Servlet programming model, `SipServlet`.

The SIP Servlet `Registrar` provides syntax and semantic checks according to RFC 3261. It is triggered on incoming SIP REGISTER messages. The SIP Servlet `Registrar`, together

with the class `Registration` decides if the received request is valid, if wildcard requests are formulated properly, checks the sequence of incoming messages and extracts information needed to be stored in the Location Service. The `Registrar` interacts with the Location Service through the interface `LocationService`. The interface provides store, update and delete methods to be utilized by the `Registrar`. Every incoming SIP REGISTER message is parsed by a newly created instance of `Registration`. During the parsing process, one to many instances of `Contact` are created and attached to the `Registration`. If the request is valid, the `Registration` object is handed over to the Location Service for further processing.

The SIP Servlet `LookupProxy` triggers on incoming SIP INVITE messages and utilizes the interface `QueryLocationService` to resolve abstract SIP addresses into concrete termination points. The interface offers only query methods to `LookupProxy`.

The implementation of `LocationService` and `QueryLocationService` is represented by the class `InMemoryLocationService`. The class organizes the storage of the information provided by the `Registrar` and manages the expiration of contacts.
The class `InMemoryLocationService` relies on the class `InMemoryLocationStore` which actually stores the information. If aiming for a failover safe and high-available solution the class `InMemoryLocationStore` needs to be exchanged by a more sophisticated approach.

A working example

The final section of the paper presents a working example where two users, Alice and Eduardo, register themselves with the SIP Registrar serving "thesipdomain.de". Afterwards, Eduardo establishes a session towards Alice.
The example shows the server log files – here a BEA WebLogic SIP Server[5] installation – and point to the actual Java classes implementing the shown functionality.
The SIP user agents utilized here in the example are the freeware "Phoner"[7] version 1.75 and "X-Lite" by Counerpath[8].

Initialization

During the start of the application server, the method `init()` of the SIP Servlet programming model is invoked.

```
…
com...Registrar --- Registrar initialized. Serving domain:
thesipdomain.de, minimum Expire time is set to: 10

com...LookupProxy --- LookupProxy initialized. Serving
domain: thesipdomain.de.
…
```

Both Servlets initialize attributes and read the configuration parameters from the deployment descriptor `sip.xml`.
`Registrar.java: 113ff`
`LookupProxy.java: 73ff`

Registration Alice

The user agent "Phoner" sends a SIP REGISTER message to the SIP Registrar and registers as the user "Alice".
According to the SIP Servlet programming model, the method `doRequest()` is invoked. The method is implemented in the `Registrar`'s base class, `DebugSipServlet` and dumps information to the server log.
`DebugSipServlet.java: 81ff, 133ff`

```
com...Registrar --- doRequest() [REGISTER]
com...Registrar --- doRequest() URI[sip:thesipdomain.de]
com...Registrar --- doRequest() From[alice
<sip:alice@thesipdomain.de>;tag=18467]
com.maretzke.sip.tools.registrar.Registrar --- doRequest()
To[sip:alice@thesipdomain.de]
com.maretzke.sip.tools.registrar.Registrar --- doRequest()
```

Afterwards, the method `doRegister()` of `Registrar` is invoked.
`Registrar.java: 164ff`
A new `Registration` object is created on the received SIP request.
`Registrar.java: 166`
The constructor of `Registration` assigns the address of record (`Registration.java: 98`), extracts the expiration times (`102ff`), identifies the contact addresses (`107ff`), looks for the parameter "q" – the relative importance of the contact binding (`115`), checks for wildcards (`118ff`), and does some semantic checks on the structure of the request (`137ff`).

```
-----------------------------------------
CONTACT == [[Mon, 16 Jul 2007 16:23:26 CEST |
sip:alice@192.168.178.200:5070, 600, -1.0 ]]
```

Afterwards, the `Registration` object dumps information about the contacts found in the SIP REGISTER message.
`Registration.java: 144`

```
com...Registrar --- locationService =
com...InMemoryLocationService@1a67700
com...Registrar --- Request domain = thesipdomain.de
com...Registrar --- Request domain is valid? - true
com...Registrar --- Request CSeq = 1
```

```
com...Registrar --- Request Call-ID = 8030A562-1432-DC11-
ADA7-005056C00008@192.168.178.200
com...Registrar --- Is the registration valid? - true
com...Registrar --- Is contact list empty? - false
com...Registrar --- Is wildcard request? - false
com...Registrar --- Registration valid? --> true
```

Back in the `doRegister()` method of the `Registrar` Servlet, further information about the processed message is written to the server log file.

`Registrar.java: 172ff`

The Registrar starts checking the validity of the request (`Registrar.java: 183ff`) and decides to proxy or process the request (`195ff`). It processes the Require header (`213ff`), checks the validity of the URI (`230ff`), unescapes the URI (`240`), calculates and verifies the minimum expiration time (`246ff`), decides to add or omit the binding (`253ff`), and eventually creates the final `200 OK` message (`277ff`).

```
------------------------------------
Bindings:
1: [AOR]: sip:alice@thesipdomain.de
callID: 8030A562-1432-DC11-ADA7-005056C00008@192.168.178.200
| CSEQ 1
Contacts:
  [Mon, 16 Jul 2007 16:23:26 CEST |
sip:alice@192.168.178.200:5070, 600, -1.0 ]

Expires:
1: [Mon, 16 Jul 2007 16:23:26 CEST |
sip:alice@192.168.178.200:5070, 600, -1.0 ]
------------------------------------
```

After the successful processing of the received SIP REGISTER message the list of bindings contains the registration of Alice. The list of bindings is dumped every 5 seconds by a thread inside `InMemoryLocationService`.

`InMemoryLocationService.java: 53ff`

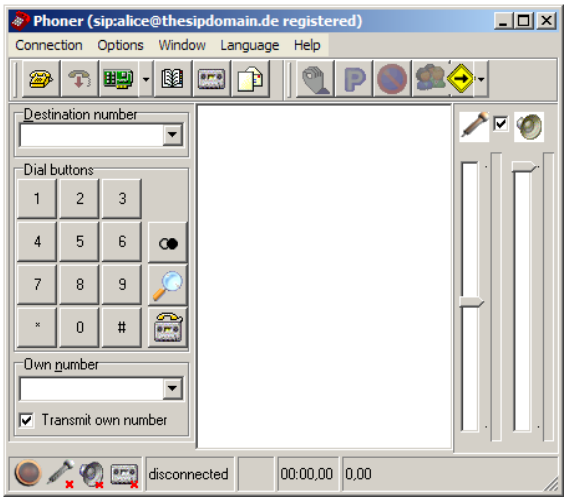

Figure 5 Phoner registered as "Alice"

Registration Eduardo

The registration process for Eduardo is essentially the same as described for Alice. A difference on user agent side is the utilization of "X-lite".

The server log file is quite comparable to the process described above.

```
------------------------------------
com...Registrar --- doRequest() [REGISTER]
com...Registrar --- doRequest() URI[sip:thesipdomain.de]
com...Registrar --- doRequest() From[Eduardo
<sip:eduardo@thesipdomain.de>;tag=1828369381]
com...Registrar --- doRequest() To[Eduardo
<sip:eduardo@thesipdomain.de>]
com...Registrar --- doRequest()
------------------------------------------------
CONTACT == [[Mon, 16 Jul 2007 16:44:16 CEST |
sip:eduardo@192.168.178.200:7060, 1800, -1.0 ]]
com...Registrar --- locationService =
com...InMemoryLocationService@1a67700
com...Registrar --- Request domain = thesipdomain.de
com...Registrar --- Request domain is valid? - true
com...Registrar --- Request CSeq = 35147
com...Registrar --- Request Call-ID =
432E8EC99CEB4944ABA68B73A872C324@thesipdomain.de
com...Registrar --- Is the registration valid? - true
com...Registrar --- Is contact list empty? - false
com...Registrar --- Is wildcard request? - false
com...Registrar --- Registration valid? --> true
------------------------------------
Bindings:
1: [AOR]: sip:alice@thesipdomain.de
callID: 8030A562-1432-DC11-ADA7-005056C00008@192.168.178.200
| CSEQ 1
Contacts:
  [Mon, 16 Jul 2007 16:23:26 CEST |
sip:alice@192.168.178.200:5070, 600, -1.0 ]

2: [AOR]: sip:eduardo@thesipdomain.de
callID: 432E8EC99CEB4944ABA68B73A872C324@thesipdomain.de |
CSEQ 35147
Contacts:
  [Mon, 16 Jul 2007 16:44:16 CEST |
sip:eduardo@192.168.178.200:7060, 1800, -1.0 ]

Expires:
1: [Mon, 16 Jul 2007 16:23:26 CEST |
sip:alice@192.168.178.200:5070, 600, -1.0 ]
2: [Mon, 16 Jul 2007 16:44:16 CEST |
sip:eduardo@192.168.178.200:7060, 1800, -1.0 ]

------------------------------------
```

The last part of the log file differs quite significantly. The server log lists the registered users for the SIP Registrar. Now after the registration of Eduardo the Registrar stores both bindings, for Alice and Eduardo.

Figure 6 X-Lite registered as "Eduardo"

Eduardo invites Alice

After the registration process is finished, Eduardo establishes a session with Alice. Therefore, he enters the SIP address "sip:alice@thesipdomain.de" into his user agent and initiates the session setup.

Figure 7 "Eduardo" establishes a session with "Alice"

```
-------------------------------------
com...LookupProxy --- doRequest() [INVITE]
com...LookupProxy --- doRequest()
URI[sip:alice@thesipdomain.de]
com...LookupProxy --- doRequest() From[Eduardo
<sip:eduardo@thesipdomain.de:7060>;tag=4011232543]
com...LookupProxy --- doRequest()
To[sip:alice@thesipdomain.de]
com...LookupProxy --- doRequest()
-------------------------------------
```

Again, according to the SIP Servlet programming model, the method `doRequest()` is invoked. As described above, the method dumps some information to the server log.

`DebugSipServlet.java: 81ff, 133ff`

```
com...LookupProxy --- LookupProxy received a SIP INVITE
message.
com...LookupProxy --- Proxying the request for
sip:alice@thesipdomain.de to sip:alice@192.168.178.200:5070
-------------------------------------
```

Afterwards, the method `doInvite()` of the `LookupProxy` Servlet is invoked and processes the incoming INVITE message.
First of all, the method retrieves a reference to the `QueryLocationService` interface implementation stored in the JNDI namespace (`LookupProxy.java: 106`). Next, the implementation queries the Location Service and tries to fetch a binding. If no binding is found, the Servlet simply returns a `404 Not Found` error message. If a binding is found, the `LookupProxy` proxies the received INVITE message towards the first contact address found in the binding (`129ff`).

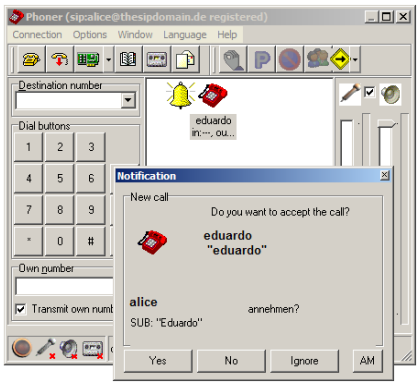

Figure 8 Incoming call from "Eduardo"

What's next?

The implementation introduced here is already quite close to the actual RFC 3261 description of a SIP Registrar. It omits only some details which might be of real importance in real-live scenarios. However, for experimental purposes the implementation should be okay already.
In two further documents the setup and the testing procedures undertaken with the SIP Registrar implementation are described in greater details.
So, what's next?
The next after this implementation might be a closer look at the issues and challenges in a distributed environment. Especially, the hurdles to overcome with distributed data storage might be very interesting for further investigation.

Important links:

SIP Registrar home & implementation download
http://www.maretzke.com/pub/howtos/sip_registrar/index.html

For questions and comments please contact me via michael@maretzke.com.

---

[1] RFC 3261 „SIP: Session Initiation Protocol",
see http://www.ietf.org/rfc/rfc3261.txt

[2] RFC 3261 – Full Copyright Statement:
*"Copyright (C) The Internet Society (2002). All Rights Reserved.
This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English."*

[3] JSR-116 „SIP Servlet API",
see http://jcp.org/en/jsr/detail?id=116
and JSR-289 "SIP Servlet v1.1",
see http://jcp.org/en/jsr/detail?id=289

[4] "commons lang" project at the Jakarta Project,
see http://jakarta.apache.org/commons/lang/

[5] BEA WebLogic SIP Server download,
see http://commerce.bea.com/showproduct.jsp?family=WLSS&major =3.0&minor=0

[6] Project Glassfish, subproject SailFin, Open Source SIP Servlet application server, see https://sailfin.dev.java.net/

[7] Phoner 1.75, see http://www.phoner.de

[8] X-Lite release 1103m build stamp 14262,
see http://www.counterpath.com/index.php?menu=Products&smenu=xlite