sip2peer Tutorial

Marco Picone

Università degli Studi di Parma
Parma, Italy

Outline

- Introduction
- Peer Creation
- Message Management
- NAT Traversal
- Conclusion & Future Work
- Contacts

Introduction

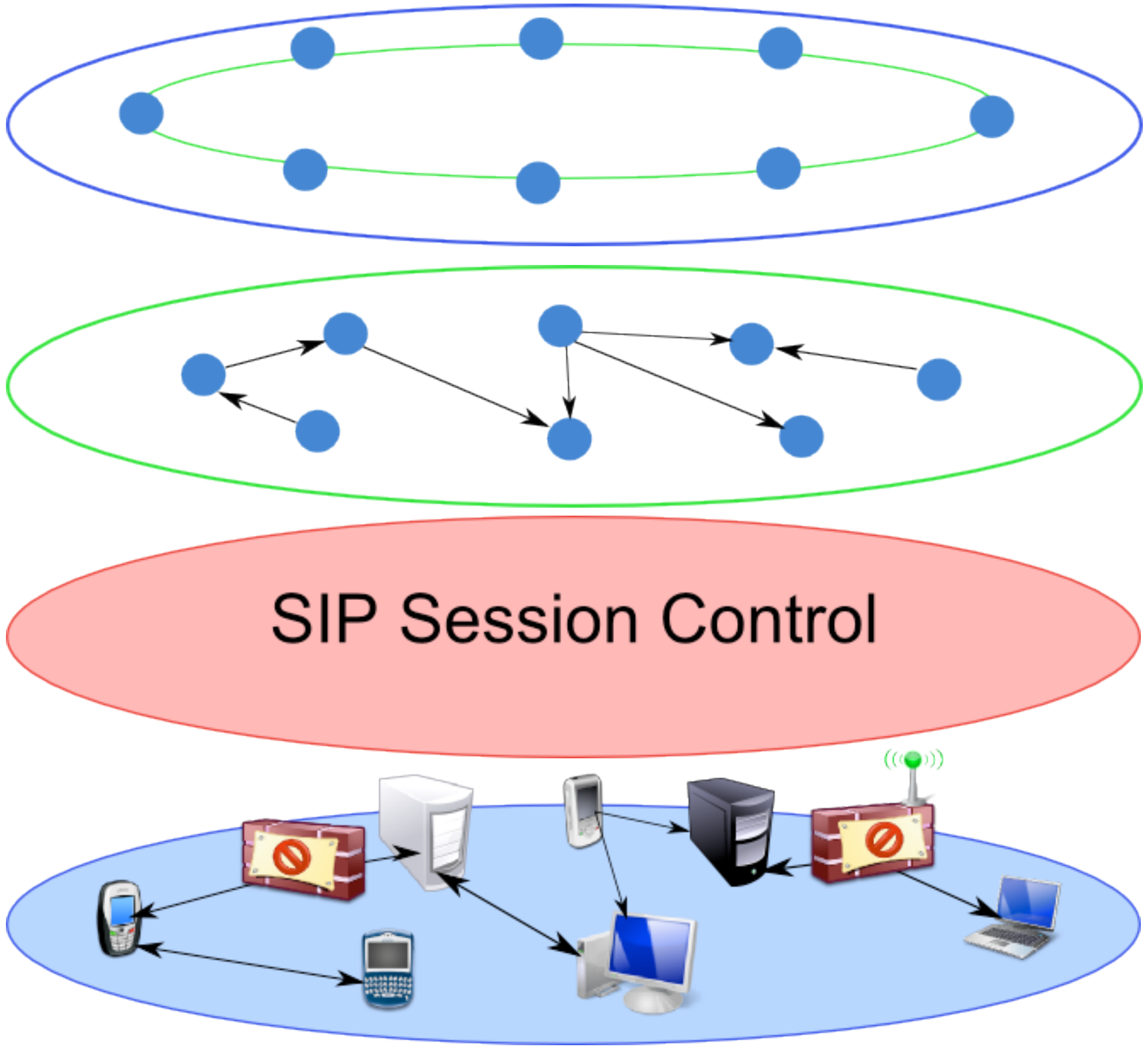
sip2peer is an open-source SIP-based middleware for the implementation of any distributed and peer-to-peer application or overlay without constrains on peer/node nature (traditional PC or mobile nodes) and specific architecture.

Main features:

- Multi-platform nature (Java & Android at the moment)
- Simple communication API with notification system
- SIP-Based platform
- NAT traversal management
- Efficient, Scalable and Configurable structure
- String and JSON based message support

"The aim is to provide a simply and scalable middleware to design and develop distributed and peer-to-peer systems allowing the developer to be focused on the overlay and the protocol, solving all issues related to communication, NAT and message exchange !"

Introduction



Overlay P2P

Middleware

Stack SIP

Stack TCP/IP

Peer Creation

To create a new peer ready to send and receive message to other active nodes, it is sufficient to extend the Peer class of sip2peer library:

```
public class SimplePeer extends Peer
```

Available constructors:

```
1) public Peer(String pathConfig, String key);

2) public Peer(String pathConfig, String key, String peerName, int
peerPort);

3) public Peer(String pathConfig, String key, String peerName, int
peerPort, BasicParser parser);

4) public Peer(String pathConfig, String key, BasicParser parser);
```

Peer Creation

All constructors accept the path of the configuration file (explained later in this presentation) and additional parameters like node key, peerName and eventually the needed message parser.

In our case we have:

```
public SimplePeer(String pathConfig) {
      super(pathConfig, "a5ds465a465a45d4s64d6a");
}
```

As mentioned sip2peer supports two message formats. It is possible to manage simple text message containing any kind of information like raw data or XML and it is also possible to natively use JSON format. Following this scalable approach, the Peer class provides all needed methods to send and receive messages allowing the developer to select the best solution according to his/her protocol and overlay.

Peer Message Listener

The following methods can be overridden and are automatically called when the node receives a message (1 for String and 2 for JSON based msgs) and when a message is correctly delivered or if there are errors.

```
1) @Override
protected void onReceivedMsg(String peerMsg, Address sender, String contentType)
{ super.onReceivedMsg(peerMsg, sender, contentType); }

2) @Override
protected void onReceivedJSONMsg(JSONObject jsonMsg, Address sender)
{ super.onReceivedJSONMsg(jsonMsg, sender); }

3) @Override
protected void onDeliveryMsgFailure(String peerMsgSended, Address
receiver,String contentType) {
}

4) @Override
protected void onDeliveryMsgSuccess(String peerMsgSended, Address
receiver,String contentType) {
}
```

Peer Message Exchange

Four important methods are inherited from the base class providing the appropriate functions to send a message to a destination node. These methods are:

```
1) public void sendMessage(Address toAddress, Address fromAddress,
String msg, String contentType)

2) public void sendMessage(Address toAddress, Address
toContactAddress, Address fromAddress, String msg, String
contentType)

3) public void send(Address toAddress, BasicMessage message);

4) public void send(Address toAddress, Address toContactAddress,
BasicMessage message)
```

Peer Message Exchange

1) and 2) do not use our JSON based message approach but allow the developer to send a generic String to the destination.

```
1) public void sendMessage(Address toAddress, Address fromAddress,
String msg, String contentType)

2) public void sendMessage(Address toAddress, Address
toContactAddress, Address fromAddress, String msg, String
contentType)
```

3) and 4) use our JSON based message approach, allowing the developer to easily extend BasicMessage class to create specific and custom messages. In this tutorial and generally in every example we will use the JSON based approach.

```
3) public void send(Address toAddress, BasicMessage message);

4) public void send(Address toAddress, Address toContactAddress,
BasicMessage message)
```

Peer Message Exchange

Methods 2 and 4 use "toContactAddress" in addition to "toAddress" parameter. The contact address represents the real destination address of the target node, while the first one is kept only as reference. This is useful in different scenarios where a peer behind a NAT advertises as contact address a specific ip:port different form the private one where it can receive incoming messages without firewall or NAT issues.

```
3) public void send(Address toAddress, BasicMessage message);

4) public void send(Address toAddress, Address toContactAddress,
BasicMessage message)
```

PeerDescriptor and PeerList Manager

The Peer class also provides the following useful and ready-to-use instances:

- **PeerDescriptor**: a simple structure to keep node information, such as:
  - name (loaded from configuration file)
  - key
  - address
  - contactAddress

- **PeerListManager**: extends Hashtable, defining a simple and ready-to-use structure to hold the peer descriptors of known nodes.

Message Definition

The base class used to define a message that could be easily transformed in a JSON string is **BaseMessage** that defines a general structure made by:

- long timestamp
- String type
- Payload payload

Payload in another base class of the library, that defines a data structure of *key/value* for the message (implements an Hashtable). The class permits different approaches to create a payload, by means of constructors and methods. It is also possible to directly add and remove parameters from it.

Detailed information is available in provided examples and in the JAVA Doc of the project.

JSON Message IMPORTANT NOTE

As already described, the transmission of BasicMessage with send methods 3) or 4) implies that the message object is converted into a JSON Object and sent as a JSON format string.

Referring to the JSON library that we are using at the moment, the JSONObject constructor uses bean getters. It reflects on all of the public methods of the object. For each of the methods with no parameters and a name starting with "get" or "is" followed by an uppercase letter, the method is invoked, and a key and the value returned from the getter method are put into the new JSONObject. The key is formed by removing the "get" or "is" prefix. If the second remaining character is not upper case, then the first character is converted to lower case.

For example, if an object has a method named "getName", and if the result of calling object.getName() is "Larry Fine", then the JSONObject will contain "name": "Larry Fine". For this reason, as shown in the JoinMessage class, it is mandatory to implement get methods to expose the parameters we want to include in the JSON object representing the message. This rule is valid not only for the Message instance, but also for each kind of Object added in the Payload.

PING Message

If for example we want to create a simple PING Message to send the node's PeerDescriptor to another active user, we just need to extend the BasicMessage class, redefining the constructor according to our needs. The result class will be:

```
public class PingMessage extends BasicMessage {

        public static final String MSG_PEER_PING="peer_ping";

        public PingMessage(PeerDescriptor peerDesc) {

          super(MSG_PEER_PING, new Payload(peerDesc));

        }
}
```

Note: PeerDescriptor class contains all needed get Methods to correctly perform the JSON Parsing of the object.

PeerList Message

static Message type definition. Used to properly manage incoming messages

```
public class PeerListMessage extends BasicMessage {

        public static final String MSG_PEER_LIST="peer_list";

        public PeerListMessage(PeerListManager peerList) {
                super(MSG_PEER_LIST, new Payload(peerList));
        }
}
```

new Payload created from a complex object (with get methods of exposed parameters)

JOIN Message

```
public class JoinMessage extends BasicMessage{

        private int numPeerList;

        public static final String MSG_PEER_JOIN="peer_join";

        public JoinMessage(PeerDescriptor peerDesc) {

                super(MSG_PEER_JOIN, new Payload(peerDesc));
                setNumPeerList(0);
        }

        public int getNumPeerList() {
                return numPeerList;
        }

        public void setNumPeerList(int numPeerList) {
                this.numPeerList = numPeerList;
        }
}
```

get method used by the JSON Parser to create the right object with Key & Value

ACK Message

It is also possible to explicitly add elements to the payload, such as in the following ACK Message:

```
public class AckMessage extends BasicMessage  {

        public static final String ACK_MSG="ack";

        public AckMessage(String status, String msg) {
                super();
                Payload payload = new Payload();
                payload.addParam("status", status);
                payload.addParam("msg", msg);
                this.setType(ACK_MSG);
                this.setPayload(payload);
        }
}
```

Free management of Payload parameters.

BluetoothDevice Message Example

```
public class BTDeviceMessage extends BasicMessage{

    public static final String TYPE="btDeviceInfo";

    public BTDeviceMessage(String btScannerId, BluetoothDeviceInfo deviceInfo) {
        super();
        Payload payload = new Payload();
        payload.getParams().put("btScannerId", btScannerId);
        payload.getParams().put("btDeviceInfo", deviceInfo);
        this.setType(TYPE);
        this.setPayload(payload);
    }
}
```

Example of structured message (extension of BasicMessage) that builds upon a basic type variable and a custom BluetoothDeviceInfo object.

BluetoothDevice Message Example

```
public class BluetoothDeviceInfo {

        public String    macAddress;
        public String   name;
        public int    RSSI;

            public int getRSSI() {
            return RSSI;
        }
        public void setRSSI(int rSSI) {
            RSSI = rSSI;
        }

    ....
```

```
...
public String getMacAddress() {
     return macAddress;
 }
 public void setMacAddress(String macAddress)
{
     this.macAddress = macAddress;
 }
 public String getName() {
     return name;
 }
 public void setName(String name) {
     this.name = name;
 }

}
```

get methods (get + VariableName) for parameters that must be parsed in JSON!

Simple Peer

In summary, what we need to code to have a full working peer, considering system.out and comments, is just:

```
public class SimplePeer extends Peer {

	public SimplePeer(String pathConfig) {
		super(pathConfig, "a5ds465a465a45d4s64d6a");
	}

	public void joinToPeer(String toAddress, String contactAddress){
		JoinMessage peerMsg = new JoinMessage(peerDescriptor);
		send(new Address(toAddress), new Address(contactAddress), peerMsg);
	}

	@Override
	protected void onReceivedJSONMsg(JSONObject jsonMsg, Address sender) {
		super.onReceivedJSONMsg(jsonMsg, sender);
		//ADD Here your incoming message management
	}

	@Override
	protected void onDeliveryMsgFailure(String peerMsgSended, Address receiver,
			String contentType) {
		System.out.println("onDeliveryMsgFailure: " + peerMsgSended);
	}

	@Override
	protected void onDeliveryMsgSuccess(String peerMsgSended, Address receiver,
			String contentType) {
		System.out.println("onDeliveryMsgSuccess: " + peerMsgSended);
	}
```

```
@Override
	protected void onReceivedMsg(String peerMsg, Address
sender,
			String contentType) {
		// TODO Auto-generated method stub
		super.onReceivedMsg(peerMsg, sender,
contentType);
		System.out.println("onReceivedMsg: " + peerMsg);
	}


public static void main(String[] args) {
		if(args.length>0){
			SimplePeer peer = new SimplePeer
("config/"+args[0]);
			System.out.println("Peer Descriptor: " +
peer.peerDescriptor);
			peer.joinToPeer(args[1], args[2]);
		}
```

Incoming Message Management

If we are using JSON feature of sip2peer library, there is an easy way to manage incoming messages according the their type.

```
@Override
protected void onReceivedJSONMsg(JSONObject peerMsg, Address sender) {

    try {

        //Get Message Type
        String msgType = peerMsg.get("type");

        //Get Message Params from the payload
        JSONObject params = peerMsg.getJSONObject("payload").getJSONObject("params");

        //Manage each kind of message
        if(msgType.equals(PingMessage.MSG_PEER_PING)){

            //Get simple String
            String name = params.get("name").toString();
            String address = params.get("address").toString()
            String key = params.get("key").toString();
            String contactAddress = params.get("contactAddress").toString();

            //If it is necessary you can directly JSON Object
            //JSONObject keyPeer = params.getJSONObject(key);
            ...........
        }

    } catch (JSONException e) {
        throw new RuntimeException(e);
    }

}
```

Peer Configuration

The Peer class provides the means to configure main parameters of the node. These values can be loaded from the file specified in the constructor, as first parameter, or in the code, by calling the appropriate methods. Tipically the most useful attributes for a node are:

- **peer_name**: name of the peer advertised in its descriptor
- **format_message**: by default is "json" but could be also "text"
- **sbc**: ip:port of the available sbc
- **keepalive_time**: keep alive time value

A configuration file has a simple structure of "key=value" with # to comment the line. A useful example:

```
peer_name=kate
#format_message=json
sbc=160.78.28.112:6067
keepalive_time=5000
```

Peer Configuration

The same configuration can be obtained in the following way:

```
SimplePeer peer = new SimplePeer("config/"+args
[0]);
peer.nodeConfig.sbc = "160.78.28.112:6067";
peer.nodeConfig.peer_name="kate";
peer.nodeConfig.keepalive_time=5000;
```

It is also possible for the developer to define his own Peer class with additional configuration parameter. For example, read the code of PeerConfig.java and its usage in FullPeer.java - both available in the example zip archive. Additional SIP configuration for the node are available only by file and the some additional field that can be used are:

- **via_addr**: contains the desired IP address for the node or the "AUTO-CONFIGURATION" to automatically select the listening interface
- **host_port**: node's port.

Peer Configuration

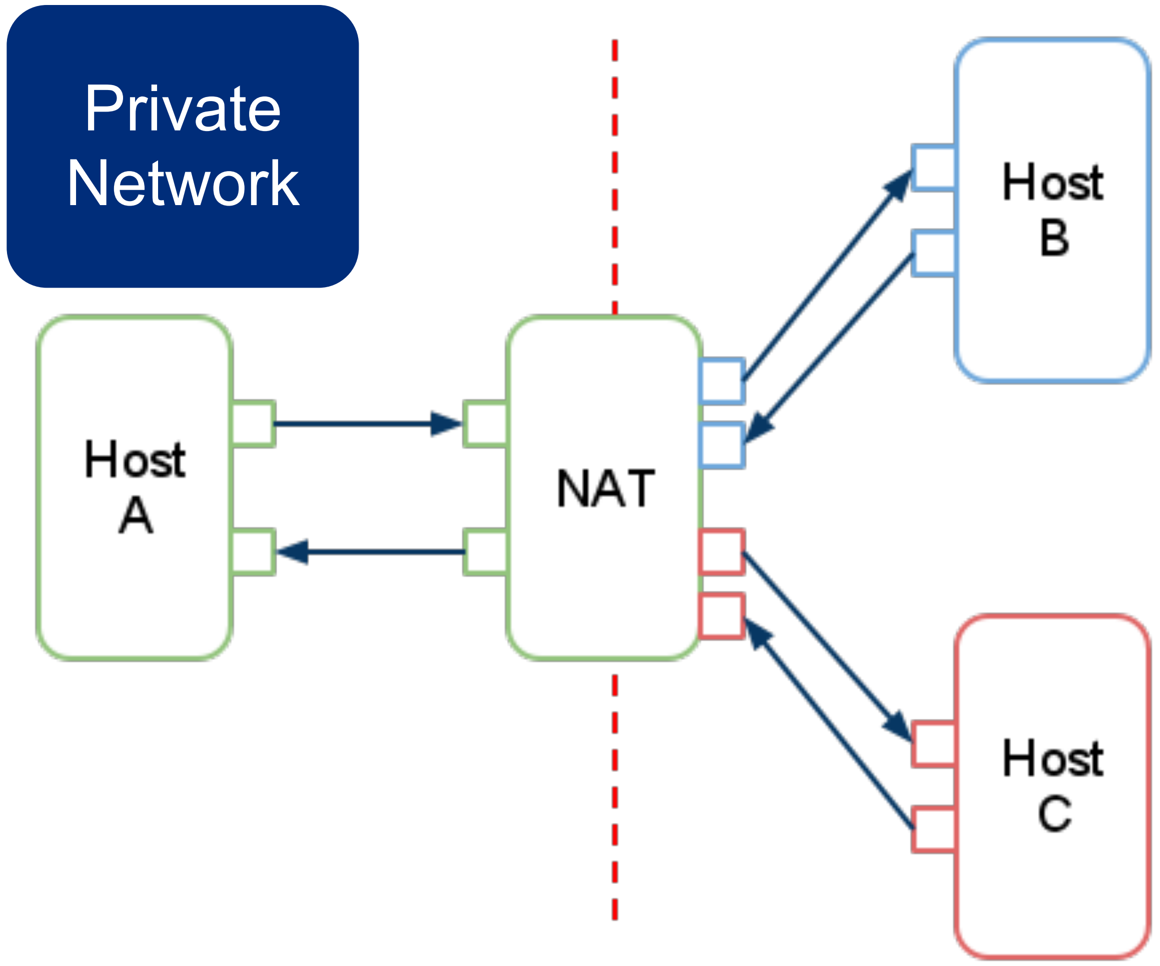
A complete configuration file with SIP parameters is for example:

```
via_addr=AUTO-CONFIGURATION
host_port=5075
peer_name=kate
#format_message=json
sbc=160.78.28.112:6067
keepalive_time=5000
debug_level=1
```

For detailed information about other SIP parameters, refer to MjSip documentation on the official website (in the Java Docs).

NAT

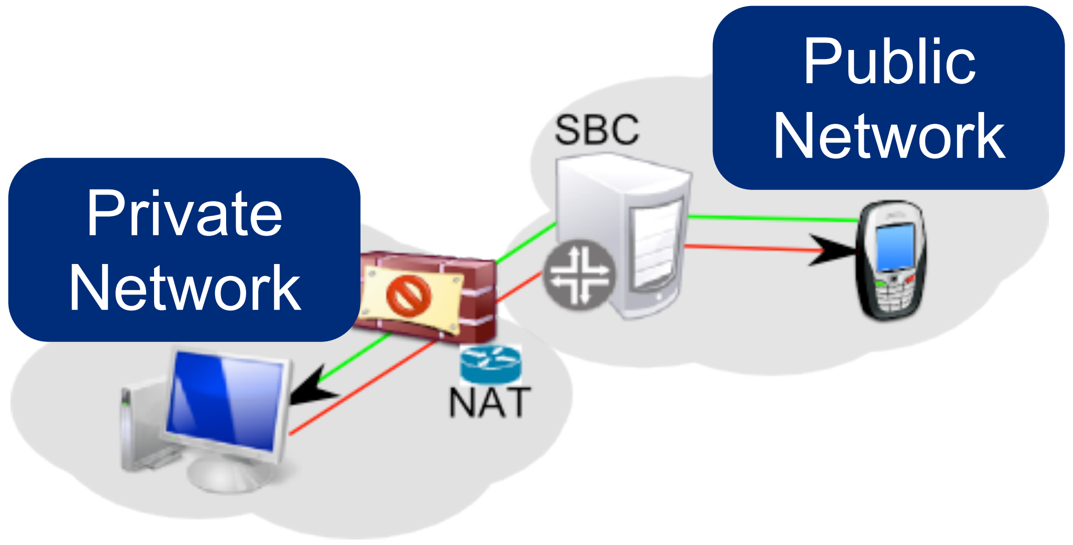
Network Address Translation (NAT) is the process of modifying network address information in datagram (IP) packet headers, while in transit across a traffic routing device, for the purpose of remapping one IP address space into another. It is a technique that hides an entire IP address space, usually consisting of private network IP addresses (RFC 1918), behind a single IP address in another, often public address space.
Nowadays NAT is a very common element in computer networking and in particular for peer-to-peer application may represent an big obstacle for the communication.



Example of symmetric NAT

NAT & SBC

In VoIP (Voice over Internet Protocol) networks, a device that is regularly deployed and used to solve NAT traversal problem is the **Session Border Controller (SBC)**. Being sip2peer based on SIP, SBC represents a natural and easily way to solve NAT traversal problems.
Shortly, we can say that in our specific case SBC is a node with public IP that allows a generic peer to check if it is behind a NAT and to request (if necessary) a public IP and port that can be used by the requesting node as contact address and that can be advertised to other peers.



The sip2peer library natively includes an SBC implementation (sip2peerSBC.zip) that can be easily configured and executed on a public IP machine.

SBC Classes

Main classes are:

- **SessionBorderController**: implements the Session Border Controller SIP Server.
- **SessionBorderControllerConfig**: class for the configuration of the Session Border Controller.
- **TestNATPeer**: class used by the SBC to check if a peer is behind NAT.
- **GatewayPeer**: implements a Gateway Peer to forward messages to the local peer. Each local peer is associated to a Gateway Peer with public address.

SBC Configuration

The main parameters for SBC configuration are:

- **via_addr**: as for node configuration
- **host_port**: as for node configuration
- **transaction_timeout**: sets the timeout for message transaction
- **test_nat_port**: listening port to test if a peer is behind a NAT
- **max_gwPeer**: Maximum number of peer managed by the SBC
- **init_port**: initialization port
- **debug_level**: as for node configuration

Example:

```
via_addr=AUTO-CONFIGURATION
host_port=6066
transaction_timeout=2000
test_nat_port=6079
max_gwPeer=15
init_port=6080
debug_level=0
```

Peer & NAT Management

In order to manage NAT on the peer side, it is only necessary to set the IP address and port of one of the available SBC servers (with public IP) and then invoke natively method provided by Peer class called **checkNAT()**.

This function starts the procedure with the SBC to verify if the requesting node is behind a NAT and if it is necessary to request a public couple IP:port. If it is the case, the library automatically requests a public address and sets it in the contactAddress (CA) parameter of the peer. Since then, the peer can advertise its CA to other peers, thus becoming reachable from the outside world.

```
public SimplePeer(String pathConfig) {
    super(pathConfig, "a5ds465a465a45d4s64d6a");
    this.checkNAT();

}
```

Peer & NAT Management

The sip2peer library provides also all methods to manually manage the interaction with the SBC. In particular, requestPublicAddress() allows to directly request a public address to the SBC, without any NAT checks and onReceive methods if override allows to intercept the communication between the node and the SBC.

```
/**
 * Request public address from Session Border Controller. Send REQUEST_PORT message to Session Border Controller
 */

public void requestPublicAddress(){...}

/**
 * When a new message is received from Session Border Controller
 *
 * @param SBCMsg message sent from SBC
 */
protected abstract void onReceivedSBCMsg(String SBCMsg);

/**
 * When the Contact Address of the peer is received from SBC by its response
 *
 * @param contactAddress
 */
protected abstract void onReceivedSBCContactAddress(Address contactAddress);
```

Conclusion & Future Work

sip2peer wants to be a *multi-platform*, *efficient*, *scalable* and *easy-to-use* library for distributed and peer-to-peer systems. In particular, the focus is on interaction and support to the new generation of mobile devices, i.e. like smartphones and tablets.

sip2peer is an **open source** project and we are really happy to share with the community our choices, design and code in order to improve it and provide new releases, examples and shared project based on sip2peer.

Future work:

- Support for new platform (iOS)
- Porting to C++
- New Examples

open source™ & Sip2Peer

License

http://www.gnu.org/licenses/gpl.html

Contact us for specific releases or additional information about the license.

Contacts

Designer(s)

- Marco Picone (picone@ce.unipr.it)
- Fabrizio Caramia (fabrizio.caramia@studenti.unipr.it)
- Michele Amoretti (michele.amoretti@unipr.it)

Developer(s)

- Fabrizio Caramia (fabrizio.caramia@studenti.unipr.it)
- Marco Picone (picone@ce.unipr.it)

http://dsg.ce.unipr.it

Contacts

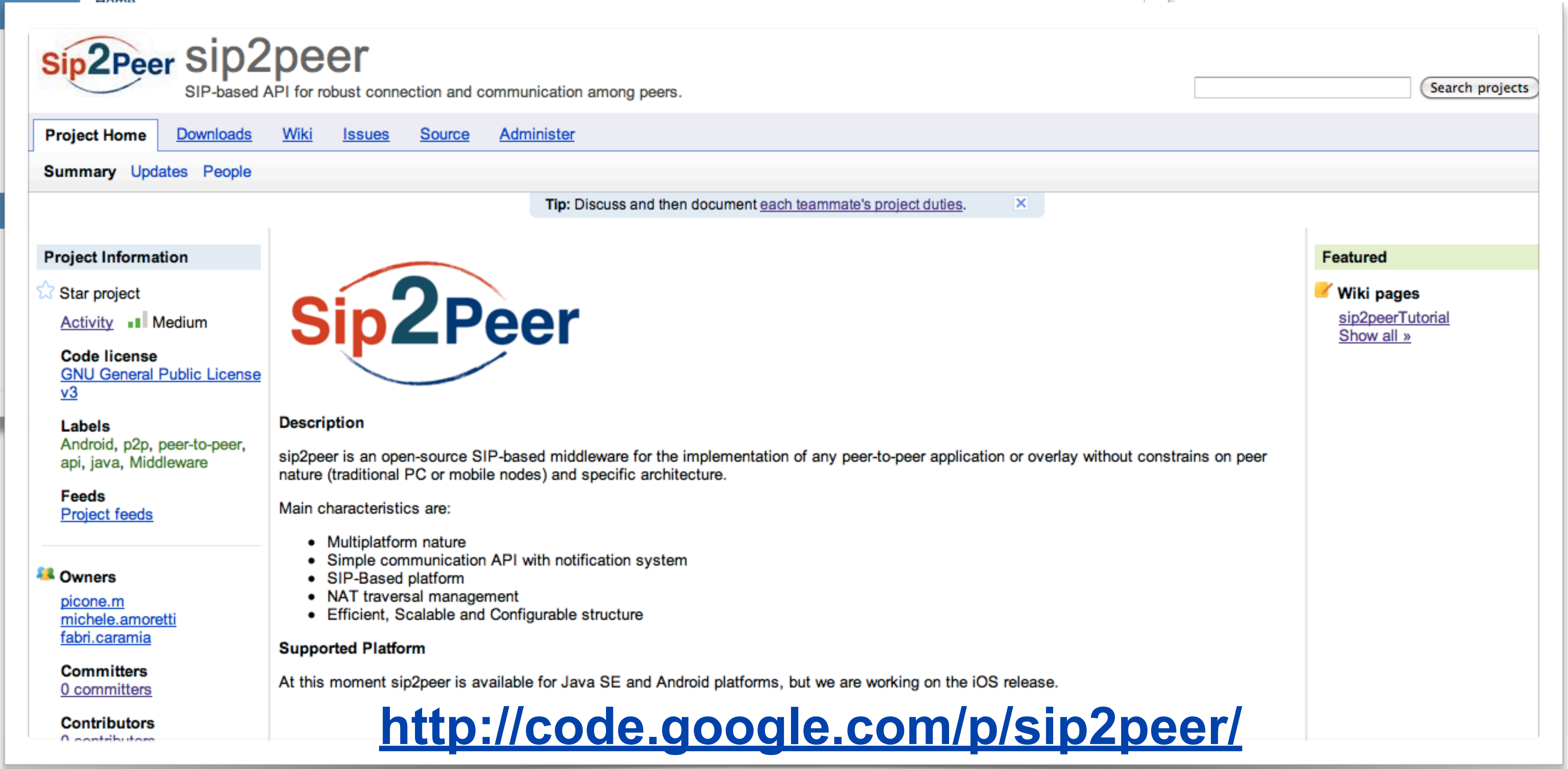
http://dsg.ce.unipr.it/?q=node/41

Distributed Systems Group
Dip. di Ingegneria dell'Informazione
Università degli Studi di Parma

About us
- DSG Home
- People
- Research
- Publications
- Students

Projects
- DSG Projects
  - DEUS
  - DGT
  - P2P & Network Coding
  - PATROL
  - sip2peer

Home



sip2peer
SIP-based API for robust connection and communication among peers.

Tip: Discuss and then document each teammate's project duties.

Project Information

Star project
Activity Medium

Code license
GNU General Public License v3

Labels
Android, p2p, peer-to-peer, api, java, Middleware

Feeds
Project feeds

Owners
picone.m
michele.amoretti
fabri.caramia

Committers
0 committers

Contributors

Description

sip2peer is an open-source SIP-based middleware for the implementation of any peer-to-peer application or overlay without constrains on peer nature (traditional PC or mobile nodes) and specific architecture.

Main characteristics are:

- Multiplatform nature
- Simple communication API with notification system
- SIP-Based platform
- NAT traversal management
- Efficient, Scalable and Configurable structure

Supported Platform

At this moment sip2peer is available for Java SE and Android platforms, but we are working on the iOS release.

Featured

Wiki pages
sip2peerTutorial
Show all »

http://code.google.com/p/sip2peer/