PeerSim HOWTO: search framework and algorithms on peersim

Gian Paolo Jesi (jesi@cs.unibo.it)
Simon Patarin (patarin@cs.unibo.it)

November, 24

1 Introduction

This tutorial is aimed to give you a step by step guide to build from scratch a new peersim application (`http://sourceforge.net/projects/peersim`). In particular, this document focuses on the implementation of a overlay search framework: on top of the basic services provided, many different kind of search algorithms can be deployed. In order to understand this tutorial, the reader is encouraged to start reading the first peersim tutorial (`http://peersim.sourceforge.net/peersim_HOWTO`) to have an idea of the basic concepts that will not be discussed any further in this document.

In this tutorial it is supposed that you and/or your workstation have:

- knowledge of O.O. programming and Java language;
- a working Java compiler ($\geq$ JDK 1.4.x);
- a working peersim source tree (you can download it from sourceforge CVS);
- the Java Expression Parser (download it from: `http://www.singularsys.com/jep`);
- isearch package distribution (available from ... cvs with anonymous login);
- (suggested) gnuplot software.

The aim of this tutorial is to be as pratical as possible.

2 Basic idea

The basic idea in the `isearch` peersim package is to provide a set of general services to develop and test P2P overlay search algorithms. In the proposed search model, it is supposed that each node has a repository of documents (a set of keys) and a query distribution (a function that maps which query to perform at each simulation cycle for each node). Each query may contains one or more keys to search for.
The overlay topology can be initialized and managed by any usual topology related peersim component (e.g.: random graphs, lattice, ...).
The framework model allows each node to initiate at most a query at each cycle and/or to forward all the messages received (stored in a buffer); each message packet can only perform one hop per cycle.
A set of tools is also present to perform some kind of time consuming tasks, such as generate different data sets representing both the key and the query distribution.
The following are the main features provided by the infrastructure:

- **common interface** for each protocol implementation which provides basic services, such as picking a neighbor, sending and forwarding a query message, provinding a common interface to the initialization process, ...;
- **initializer** component to bootstrap the data sets (storage and query). The data can be extracted by traces or generated by some distribution functions;
- **observer** component to collect data experiments and to track statistics (e.g.: query hits, duplicate packets, ...);
- **packet component** to model the actual message exchange between peers; each message has a unique identifier and contains informations about the sender, the number of hops and stores the actual query keys.

2.1 Services

The basic services are the foundation to build the search protocols. An abstract class `SearchProtocol` provides a first common implementation of those services; all `isearch` protocols are supposed to inherit from this class. Usually, each protocol is made of two distinct parts: *active* and *passive*; the former represents the pro-active behaviour (e.g.: nodes injects queries into the system according to a predefined distribution), and the latter reacts to the incoming messages according to the specific protocol behaviour.
The abstraction level provided to the user is quite high: `Nodes` (a peersim specific type to represent hosts), `SMessages` (see 2.2) packet type and primitive such as *send* and *forward* are the common metaphores used. Please

note that the usual primitive *receive* is not explicitly needed, but the message reception is handled transparently by the platform as follows: when a peer sends a packet to a neighbor, the message is stored in the neighbor node buffer (`incomingQueue` structure) and a predefined protocol behaviour pops messages from the buffer and handles them according to some strategy. Then, the control is passed to the user defined protocol behaviour.
The data structures needed by each node in the infrastructure are the following:

- **messageTable**: is a hashtable that maps a packet to an integer; it represents the number of times the current node has seen this packet before.
- **hitTable**: is a hashset that stores the packets for which the current node reports a query hit.
- **incomingQueue**: is a list that buffers the incoming packets; it is accessed in a FIFO fashion.
- **view**: the current node neighbor list view. It is managed by the Linkable interface methods.
- **keyStorage**: is a hashmap mapping a key to an integer; the latter represents the requency of the key.
- **queryDistro**: is a treeset mapping integers to key array; the former represents the cycle in which scheduling the query and the latter represents the packet query payload.

The `SearchProtocol` class has to implement two usual peersim interfaces: `peersim.cdsim.CDProtocol` and `peersim.core.Linkable`. In brief, the former handles the passive behaviour (in the `nextCycle()` method) and then invokes the abstract method `process()` and the user protocol must provide an implementation for it. The latter handles the a logical reference to the actual topology that can be initialized by any topology related peersim component.
The code to handle the passive behaviour is the following:

```
public abstract class SearchProtocol implements CDProtocol, Linkable {
...
// interface CDProtocol:
public void nextCycle(Node node, int protocolID) {
    int currentTime = CommonState.getT();

    Iterator iter = incomingQueue.iterator();
    while (iter.hasNext()) {
```

```
        SMessage mes = (SMessage) iter.next();
        if (mes.hops == (currentTime - mes.start + 1))
             continue;
        Integer actual = (Integer) this.messageTable.get(mes);
        int index = (actual != null ? actual.intValue() + 1 : 1);
        this.messageTable.put(mes, new Integer(index));
        this.process(mes);
        iter.remove();
    }
}

public abstract void process(SMessage mes);
```

The current node buffer (`incomingQueue`) is visited and for each element (message) a check about the packet age is performed and, if it is acceptable, the message query is processed by the protocol specific policy implemented in the `process()` method and then the message is removed from the buffer. Note that the packets are inserted in the `messageTable` structure to be useful for statistics.
The code for `send()` and `receive()` primitive is the following:

```
public void send(Node n, SMessage mes) {
    try {
         SMessage copy = (SMessage) mes.clone();
         copy.hops++;
         this.messageTable.put(mes, new Integer(1));
         SearchProtocol sp = (SearchProtocol) n.getProtocol(pid);
         sp.incomingQueue.add(copy);
    } catch (CloneNotSupportedException ex) {
         ex.printStackTrace();
    }
}

public void forward(Node n, SMessage mes) {
    if (mes.hops < ttl) {
         try {
             SMessage copy = (SMessage) mes.clone();
             copy.hops++;
             SearchProtocol sp = (SearchProtocol) n.getProtocol(pid);
             copy.type = SMessage.FWD; // sets FWD type
             sp.incomingQueue.add(copy);
         } catch (CloneNotSupportedException cnse) {
             System.out.println("Troubles with message cloning...!");
```

```
        }
    }
}
```

The first method is used by the active behaviour: when new queries are genereted and injected into the system, while the second one is used by the other node to propagate the query to their neighbor according to some strategy.
Note that the messages are always cloned before been moved around. The send function basically increments the packet age by one and sends it to a neighbor buffer. The forward function performs quite the same operation, but first it checks if the packet age is acceptable, otherwise it discards the message.

The framework exposes a simple interface to be accessed during the initialization phase. Two methods are available for adding the query data and the keys into the storage.

```
public void addQueryData(int cycle, int[] keys) {
    this.queryDistro.put(new Integer(cycle), (Object) keys);
}

public void addKeyStorage(Map entry) {
    this.keyStorage.putAll(entry);
}
```

A very flexible way to verify the hit of a query is provided by the `matches()` method. It verifies, one by one, all the keys inside a message query with the storage keys and return an array consisting of matching keys (clearly the array may be empty).

```
protected int[] matches(int[] keys) {
    int[] result = null;
    ArrayList temp = new ArrayList();
    for (int i = 0; i < keys.length; i++) {
        if (this.keyStorage.containsKey(new Integer(keys[i]))) {
            temp.add(new Integer(keys[i]));
        }
    }
    if (temp.size() > 0) {
        result = new int[temp.size()];
        for (int i = 0; i < temp.size(); i++) {
            result[i] = ((Integer) temp.get(i)).intValue();
```

```
        }
    }
    return result;
}
```

Another comparison method is also provided and the following is its signature:

```
protected boolean match(int[] keys)
```

It returns a success or unsuccess operation status acoording to the defined comparison method. The comparison method is configurable by setting a parameter (`and_keys`) in the configuration file. It allows the user to choose between a matching OR / AND behaviour to compare keys.

2.2 Packets (SMessage)

The decision to explicitely model packets is due to maximize the framework modularity. Packets store important parameters such as:

- the originator node of the query;
- a unique sequence number to distinguish packets;
- the packet creation time in term of cycle;
- the packet age in terms of *hops*;
- the message type (e.g.: query, forrward, ...);
- the payload (an array of keys);

The SMessage class has to implement some Java specific intefaces to support the cloning process performed by the communication primitives and to be efficiently distinguishable and to be collected in a hashtable like structure. The cloning features requires the implementation of the `java.lang.Cloneable` interface and the implementation of a uitable `clone()` method; the other feature istead, requires the override of the following `java.lang.Object` class methods: `hashCode()` and `equals()`.

```
public class SMessage implements Cloneable {
    public static final int QRY = 0;
    public static final int FWD = 1;
    public static final int HIT = 2;

    private static int seq_generator = 0;
```

```
    public int hops, type, seq, start;
    public Node originator; // the query producer
    public int[] payload; // an array of keys

    public SMessage(Node originator, int type, int hops, int[] payload) {
    ...
    }

    public Object clone() throws CloneNotSupportedException {
        SMessage m = (SMessage) super.clone();
        return m;
    }

    public int hashCode() {
        return seq;
    }

    public boolean equals(Object obj) {
        return (obj instanceof SMessage) && (((SMessage) obj).
    seq == this.seq);
    }
...
}
```

2.3 Statistics

The generation of the query statistic is made by a peersim observer (`peersim.reports.Observer`) interface object. At the end of each cycle the observer runs and collects with global knowledge the data about the packets stored at each node and generates statistics. At the end of each cycle or at the end of the whole simulation, the observer generates the following data touples about query packets:

$$\begin{pmatrix} queryID \\ message\ TTL \\ number\ of\ times\ the\ packets\ has\ been\ seen \\ number\ of\ successful\ packet\ hits \\ total\ number\ of\ messages\ sent\ for\ this\ query \end{pmatrix}$$

3 Protocols on top of the framework

The set of services provided by the infrastructure allows the fast creation of search like protocols. A basic random walk implementation approach is presented in the following pages.

3.1 Random walk

In a random walk, when a node receives a message it simply forwards the packet to a random chosen neighbor after having compared the message query with its local keys repository.

```
public class RWProtocol extends SearchProtocol {
    /** Parameter for the number of walkers at the query
     *initiation. It must be < then view size.  Default is 1.
     */
    public static final String PAR_WALKERS = "walkers";
    protected int walkers;

    public RWProtocol(String prefix, Object obj) {
        super(prefix, obj);
        int match= peersim.config.Configuration.getInt(prefix+"."+PAR_ANDMATCH, 0);
        if (match == 1 ) this.andMatch = true;
        else this.andMatch = false;
    }

    public void process(SMessage mes) { // "Passive" behaviour implementation
        // checks for hits and notifies originator if any:
        boolean match = this.match(mes.payload);
        if (match) this.notifyOriginator(mes);

        // forwards the message to a random neighbor:
        Node neighbor =
            (Node) this.view.get(CommonRandom.r.nextInt(view.size()));
        this.forward(neighbor, mes);
    }

    // "active" behaviour implementation: makes query
    public void nextCycle(peersim.core.Node node, int protocolID) {
        super.nextCycle(node, protocolID);
        // this will handle incoming messages

        int[] data = this.pickQueryData(); // if we have to produce a query...
        if (data != null) {
            SMessage m = new SMessage(node, SMessage.QRY, 0, data);
            for (int i = 0; i < this.walkers && i < this.view.size() ; i++) {
                this.send((Node) this.view.get(i), m);
            }
        }
    }
}
```

The `nextCycle()` method is overridden, but it first calls its superclass implementation (`super.nextCycle()`) to process the incoming buffered messages; then if there are any queries scheduled for the current cycle, it starts the queries sending messages to the neighbors. The number of neighbor to send the message to is defined by the `walkers` configurable parameters, shown in the first code lines.
The behaviour with which the buffer packets ae handles is defined by the `process()` method implementation. First a check about the key match is performed and a suitable data structure is updated accordingly, then a new neighbor is choosen and a the message in forwarded. If the message age is too large, then the message will be automatically discarded.

3.2 Restricted versions

Each protocol can be easily converted to a *restricted* version. The restriction is made on the strategy with which a neighbor is choosen. The node neighbors are probed to find a candidate that has never seen before the message that is going to be sent. If there isn't such candidate, then a random node is choosen as usual.
The code is the following and extends the standard version:

```
public class RRWProtocol extends RWProtocol {
    public RRWProtocol(String prefix, Object obj)  {
        super(prefix, obj);
    }

    public void process(SMessage mes) {
        // checks for hits and notifies originator if any:
        boolean match = this.match(mes.payload);
        if (match) this.notifyOriginator(mes);

        // forwards the message to a random FREE neighbor:
        Node neighbor = this.selectFreeNeighbor(mes);
        this.forward(neighbor, mes);
    }
}
```

The only difference is the way the `process()` method adopts to forward the messages. A specific function (`selectFreeNeighbor()` part of the basic services provided) support the restricted protocol version and chooses a neighbor according to this strategy.

4 Usage examples

To run a search protocol in the isearch package, a suitable peersim configuration file is needed. The following configuration should be sufficient to start using the framework:

```
1  # PEERSIM EXAMPLE iSEARCH
2  random.seed 1234567890
3  simulation.cycles 200
4  simulation.shuffle
5
6  overlay.size 10000
7
8  protocol.0 example.isearch.RWProtocol
9  protocol.0.ttl 100
10 protocol.0.walker 1
11
12 init.0 peersim.dynamics.WireRegularRandom
13 init.0.protocol 0
14 init.0.degree 10
15
16 init.1 example.isearch.SearchDataInitializer
17 init.1.protocol 0
17 init.1.keywords 10000
19 init.1.query_nodes 100
20 init.1.query_interval 1
21 init.1.max_queries 1
22 init.1.and_keys 0
23
24 observer.0 example.isearch.SearchObserver
25 observer.0.protocol 0
26 observer.0.verbosity 0
```

The **lines from 2 to 6** regard the global peersim configuration, such as the seed for the random number generator, the number of simulation cycles to perform, the node shuffle switch (to avoid picking nodes in the same order at each cycle) and the network size.
The only defined protocol is the random walk protocol (**lines 8, 10**) and the `ttl` parameter defines the maximum allowed age for packets.
The first initializer (**lines from 12 to 14**) defines a random graph topology over the nodes, and the degree is user definible.
The second initializer (**lines from 16 to 22**) defines the most important aspect: how the key repository and the query distribution are initialized at each node. The following aspects are customizable:

- **protocol**: the protocol to initialize (the random walk in this case);
- **keywords**: determine the number of distinct keywords in the system;
- **query_nodes**: determine the number of nodes emitting queries (default is network size);
- **query_interval**: used to determine the average time interval (in cycles) between queries for the Poisson distribution (default is 10);
- **max_queries**: determinates the maximum number of queries emitted by a single node (default is unlimited).
- **and_keys**: the strategy with which keys are compared (AND, OR behaviour). The default is OR strategy.

Finally, the observer is defined from **line 24 to 26**. The only two parameters defines the protocol to inspect and how to produce data (cycle by cycle or at the end).

To run the config file, assuming the peersim classes presence in the CLASSPATH, just type:

```
java peersim.Simulator <path-to-configfile>/config-isearch.txt
```