PeerSim HOWTO: iSearch framework and algorithms  
(updated edition)

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

January 2, 2008

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: iSearch. On top of the basic services provided, many different kind of search algorithms can be implemented. In order to understand this tutorial, the reader is encouraged to read at least the first peersim tutorial (http://peersim.sourceforge.net/peersimHOWTO) to have an idea of the basic concepts that will not be discussed any further in this document. Another help source about iSearch and Peersim in general is the peersim help forum (http://sourceforge.net/forum/forum.php?forum_id=315476) and we suggest to check it before asking questions.

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

- knowledge of O.O. programming and Java language;
- a working JDK 5 installation (or better);
- a working peersim installation (you can download it from sourceforge: http://peersim.sourceforge.net/#download);
- the Java Expression Parser version $\geq$ 2.3.0 (download it from: http://www.singularsys.com/jep);
- the iSearch package is one of the packages available from the extra repository of Peersim by typing:
  
  # cvs -d:pserver:anonymous@peersim.cvs.sourceforge.net:/cvsroot/peersim login
  
  # cvs -z3 -d:pserver:anonymous@peersim.cvs.sourceforge.net:/cvsroot/peersim co -P extras
The aim of this tutorial is to be as practical 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 contain one or more keys to search for.

iSearch represents a simple, but limited way to model the actual message exchange among nodes using the cycle driven simulation environment of peersim. However, this feature is very inefficient as it needs lots of memory space and it is not scalable. When large scale simulations are involved and to use the full potential of peersim, use the event simulation model.

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, providing 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 information 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: Node (a peersim specific type to represent hosts), SMessage (see Section 3), packet type and primitive such as send and forward are the common metaphors 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 behavior 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
- **routingTable**: stores to whom a message has been sent and how many copies
- **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 behavior is the following:
public abstract class SearchProtocol implements CDProtocol, Linkable {

    // interface CDProtocol:
    public void nextCycle(Node node, int protocolID) {
        int currentTime = CDState.getCycle();

        Iterator iter = incomingQueue.iterator();

        while (iter.hasNext()) {
            SMessage mes = (SMessage) iter.next();
            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) 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));
            updateRoutingTable(n, mes);
            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);
                updateRoutingTable(n, mes);
                sp.incomingQueue.add(copy);
            }
        }
    }

    4
The first method is used by the active behaviour: when new queries are generated 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.

```java
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).

```java
protected int[] matches(int[] keys) {
    int[] result = new int[keys.length];
    ArrayList<Integer> temp = new ArrayList<Integer>(keys.length);
    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] = temp.get(i);
        }
    }
    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 according 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. The actual behavior is managed by the flag parameter and_key of the SearchProtocol class; the default behavior is OR strategy.

3 Packets (SMessage type)

The decision to explicitly 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, forward, ...)
- the payload (an array of keys)

The SMessage class has to implement some Java specific interfaces 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 suitable clone() method; the other feature instead, requires the override of the following java.lang.Object class methods: hashCode() and equals().

```java
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);
}
...

3.1 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 tuples
about query packets:

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

4 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.

5 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;
/** Creates a new instance of RWProtocol */
public RWProtocol(String prefix) {
    super(prefix);
    walkers = Configuration.getInt(prefix + "." + PAR_WALKERS, 1);
}

/** "Passive" behaviour implementation: process key similarity
* and notifies any match and forwards messages. */
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 neighbor:
    Node neighbor = this.getRNDNeighbor();
    this.forward(neighbor, mes);
}

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

    // if we have to produce a query...
    int[] data = this.pickQueryData();
    if (data != null) {
        System.err.println("DATA");
        SMessage m = new SMessage(node, SMessage.QRY, 0, data);
        // produces the specified number of walkers:
        for (int i = 0; i < this.walkers && i < this.degree(); i++) {
            this.send((Node) this.getNeighbor(i), m);
        }
    }
}

The nextCycle() method is overridden, but it first calls its superclass
implementation (super.nextCycle()) to process the incoming buffered mes-
sages; then if there are any queries scheduled for the current cycle, it starts
to send the query messages to the neighbors. The number of neighbors
to send the message to is defined by the walkers configurable parameters,
shown in the first code lines.

The behavior with which the buffer packets are 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 the message in forwarded. If the message is too old,
then the it will be automatically discarded.

6 Restricted versions

Each protocol can be easily converted to a restricted version. The restriction is made on the strategy with which a neighbor is chosen. 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 chosen as usual. The code is the following and extends the standard version:

```java
public class RRWProtocol extends RWProtocol {
    /** Constructor */
    public RRWProtocol(String prefix) {
        super(prefix);
    }

    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 (i.e., `selectFreeNeighbor()` part of the basic services provided) supports the restricted protocol version and chooses a neighbor according to this strategy.

7 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:

```
# PEERSIM EXAMPLE iSEARCH
1 random.seed 1234567890
2 simulation.cycles 50
3 control.0 peersim.cdsim.Shuffle
4
5 network.size 100
```
# The iSeach protocol instance:
protocol.search isearch.RWProtocol
protocol.search.ttl 20
#protocol.search.walkers 2

init.0 peersim.dynamics.WireKOut
init.0.protocol search
init.0.k 20

init.1 isearch.SearchDataInitializer
init.1.protocol search
init.1.keywords 1000
init.1.query_nodes 1
init.1.query_interval 1
init.1.max_queries 1

control.0 isearch.SearchObserver
control.0.protocol search
control.0.verbosity 1

control.1 peersim.reports.DegreeStats
control.1.protocol search
control.1.method list

The lines from 1 to 7 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, 11) and the ttl parameter defines the maximum allowed age for packets. Optionally, the walkers parameter can be set (default value: 1 walker).

The first initializer (lines from 13 to 15) defines a random graph topology over the nodes, and the degree is user definable (e.i., k parameter). The second initializer (lines from 17 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 during the simulation (default: all nodes);
• **query_interval**: used to determine the average time interval (in cycles) between queries for the Poisson distribution (default is 10);

• **max_queries**: determines the maximum number of queries emitted by a single node (default is unlimited).

Finally, the observer is defined from line 24 to 26. The two parameters define respectively the protocol to inspect and how to show data (cycle by cycle or at the end). To run the config file, assuming that the CLASSPATH system variable has been set correctly, just type:

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