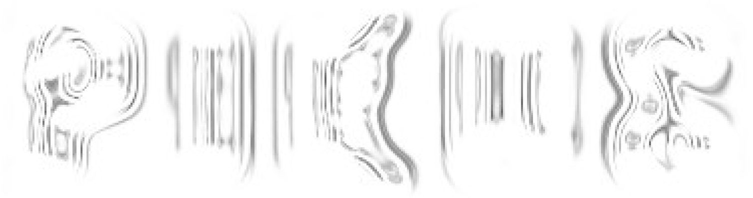
Olso net Communications

SLOTS
A Sampling-based Location Tracking System for Independent Living Facilities



Version 1.0

July 16, 2013

1 Preamble

This document describes software for experimenting with profiling-based, indoor location tracking systems built of Olsonet wireless nodes. The only hardware prerequisite assumed from the RF module is its ability to report some measure of RSS (Received Signal Strength) indication of a packet arriving from a node in the neighborhood.

This is a complete re-write of our previous, tentative, coordinate-based location tracking software which we have been playing with for a while with mixed success. That system was devised solely as an experimental tool and its practical applicability was limited. The main problem was its orientation for solving geometric problems in Cartesian plane using <x,y> coordinates both as the input (for profiling) and the output (in responses to queries). In practical positioning applications we may deal with layouts where:

1. Exact Cartesian coordinates are difficult to obtain. The user may find the procedure tedious, and it may be unnatural for him to think in terms of coordinates.
2. The layout is not planar; it need not even be nicely structured, say, as a straightforward pile of planar floors (switching to 3D coordinates may not be very helpful from the practical point of view).
3. Cartesian coordinates may not be the most natural (expected) output from the system.

Point 3 is most significant. For the class of our targeted location-tracking applications, the most natural output would be "an area" understood as a named place where one can go to find the tracked object/person. In particular, the required accuracy of location estimation (expressed in meters) is not a fixed parameter of the system (which comes natural in a generic coordinate-based approach), but it depends (possibly quite strongly) on the contextual characteristics of the "area". For example, the granularity of a room (or even a flat) may be quite satisfactory in some circumstances, while it may be necessary to accurately tell apart adjoining rooms or flats.

In this light, it seems a bad idea to operate in terms of Cartesian coordinates first trying to pinpoint the abstract Cartesian location of the tracked object (with some coordinate-oriented algorithm), and then transforming that location into some context-specific descriptive representation (in terms of areas of interest). A more natural alternative to this two-step conversion approach would be to forgo the coordinates altogether and base the algorithm solely on symbolic (descriptive) locations (because they are all that matters).

2 The principle of operation

A deployed location tracking system consists of a number of static *Pegs* collecting RF signals from mobile *Tags* whose positions are being tracked. The tracking is driven by a database of *readings*, which represent signal strength (RSS) samples previously collected from the interesting locations within the perimeter of the monitored area. The geographic relationships between the locations is irrelevant. In particular, the system assumes nothing about the dimensionality of the problem (2D versus 3D).

Before the deployment, the interesting locations should be identified and labeled (named) and the area should be profiled. By that we mean collecting samples (readings) from the labeled locations. Note that we do not care about any other locations: any reading stored in the database must be attributed to one of the agreed-upon labeled locations. More specifically, the profiling consists in sending signals from Tags from within the labeled locations and storing the strengths of those signals (RSS) as perceived by those Pegs that can receive them.

A location-tracking problem is presented to the system as a query consisting of a number of readings obtained by the Pegs from a tracked Tag. That query is matched to the samples stored in the database to produce an answer which always boils down to one of the labeled locations. Generally, we do not care about answers like “between locations A and B”. We want a reasonably good guess as to the actual single location where the tracked Tag is likely to be found. In other words, we assume that the labeled locations exhaustively cover the entire area of interest and they never intersect pairwise. In some situations, e.g., when several locations are naturally connected and accessible from one another, the non-intersection postulate can be relaxed (a positioning error within the set of such locations may be easy to tolerate).

The present system consists of three components:

1. The database storing the profile samples as well as information about locations. This is a standard SQL (SQLite 3) database.
2. The location server, i.e., a daemon accepting location queries (over TCP/IP socket connections) and sending replies to those queries.
3. The maintenance program, i.e., a Tcl script for viewing and modifying the database. At present, the script provides the only way to augment the database with new samples.

The isolation of the database from the server makes it easier to delegate maintenance (less time-critical than the location service) to scripts. Also, data can be entered into the database (and inspected) manually (with reasonable ease) via standard command-line tools, e.g., the sqlite3 program.

2.1 The database

There are just three tables defined with the following SQL statements (when the database is initialized):

```
CREATE TABLE locations (
  name TEXT PRIMARY KEY NOT NULL,
  lid INTEGER UNIQUE NOT NULL,
  comment TEXT
);
CREATE TABLE samples (
  sid INTEGER PRIMARY KEY,
  lid INT NOT NULL,
  tag INT NOT NULL,
  gf INT NOT NULL,
  lf INT NOT NULL
);
CREATE TABLE readings (
  sid INT NOT NULL,
  peg INT NOT NULL,
  rss INT NOT NULL,
  PRIMARY KEY (sid, peg)
);
```

The *locations* table maps integer location identifiers (attribute *lid*) to descriptive strings (*name*) and (optional) even more descriptive *comments*. This table is not used by the location server which deals exclusively with the integer identifiers of locations.

The complete description of a sample is split between the remaining two tables. This is to make sure that different samples may consist of different numbers of readings; also, that a single reading can easily be accessed as an independent entity and, possibly, removed or replaced. Creative “massaging” of the samples (after the profiling) will likely be needed to bring the accuracy of location tracking to the required level.

A sample (sans the actual readings) is described by these attributes:

sid This is an internally generated, unique, 64-bit, integer identifier of the

	sample also used as the PRIMARY KEY of the *samples* table. Note that readings are matched to samples by this attribute.
lid	This is the numerical identifier of the location from which the sample was taken.
tag	The identifier of the node (Tag) responsible for generating the sample. This attribute is not used by the present algorithm, but it is made available for future extensions.
gf	Global features. This is a set of descriptors (packed into a 32-bit integer value) pertaining to the global characteristics of the RF signal, e.g., the channel number or the antenna type.
lf	Local features, i.e., the local characteristics of the sample; for example, the orientation of the Tag or its (quantized) distance from the floor (e.g., standing, sitting, lying).

A reading consists of these items:

sid	The identifier of the sample to which the reading belongs. A reading is deemed to belong to a given sample, if its *sid* attribute coincides with that of the sample.
peg	The identifier of the Peg node that has received the reading. Note that the pair (*sid*, *peg*) constitutes the PRIMARY KEY of *readings*, so at most one reading for a given Peg can belong to one sample.
rss	The RSS value of the signal.

The role of *gf* (global features) is to differentiate among samples collected under incompatible conditions, e.g., transmission power, frequency, and so on. We mean here some agreed-upon (and simple to express) parameters that may affect the RSS readings, which are known both at the time of collecting a sample as well as at the time of issuing a query. The idea is that the queries issued under specific conditions should only be interpreted within the context of samples (readings) collected under the same conditions. As of now, we do not have yet a standard interpretation of *gf*,[1] but we should accommodate at least the transmission power and the channel/frequency, and probably a few more, including the rough node type (C1100, CC430), and the antenna type.[2]

On the other hand, *lf* (local features) corresponds to some agreed-upon (and simple to express) parameters that may influence the RSS readings, which are known at the time of collecting the sample, but may not be known at the time of issuing a query. One characteristic of this kind may be the orientation of the Tag or its (quantized) distance from the floor.

Technically, the difference between *gf* and *lf* is that the former arrives in a query, so it can be used to preselect samples, while the latter does not (although it is available at the time of collecting a sample).

[1]Note that the server doesn't care about that interpretation. The value of *gf* is only used for matching samples to queries. Similarly, *lf* is only used for grouping samples into sets.

[2]For example, it may turn out that trimming the antenna (i.e., intentionally reducing the signal strength at the transmitter) is helpful.

Value 0 of *rss* (in a sample reading) is special (an actual RSS reading can never be zero) and marks the entry as a dummy self-profiled reading. By self-profiling we mean using the Pegs to generate samples from the locations at which they are deployed. Note that the RSS reading at the Peg triggering such a sample is unknown, but we would expect it to be extremely high (at the level of the maximum possible RSS). Formally, in such a case, the *tag* attribute of the sample should be equal to the *peg* attribute of the reading, but that is never checked. So the special value 0 can be used as a substitute for an unknown but very high signal level, e.g., indicating to the matching algorithm that a very strong signal at this Peg is required for a good match to the sample.

2.2 The algorithm

This algorithm is rather heavily parametrized, which means that it is largely generic. The parameters are supplied in an XML data file to the server.

A query looks like this:

$$\langle tag, gf, peg_1, rss_1, \ldots, peg_n, rss_n \rangle$$

where *tag* is the identifier of the Tag node generating the signal, *gf* is the set of global features used by the Tag, and the remaining attributes are the RSS readings collected at the respective Pegs.

Note that an alternative representation of the readings vector would be to have a reserved position for every Peg, thus eliminating the need for Peg identifiers, and use a special “absent” value to mark those Pegs that didn't receive the signal. Arguably, in small and close deployments, that would simplify the algorithm as well as the database design (because all samples would be the same length).

In the first step it is checked whether *n* (the number of readings) is not less than the minimum required (the *minimum* attribute of <*selection*>*,* see Section 3.3), in which case the query is rejected. If the number of readings is large enough, the algorithm preselects samples from the database to be matched to the query. The query readings are first scanned to find the Peg with the largest RSS value. Call that Peg P_m and its RSS value R_m. Then the samples selected from the database fulfill these criteria:

1. The *gf* attribute of the sample matches the *gf* attribute of the query.
2. The set of readings of the sample includes P_m and the RSS value of that reading is either 0 (meaning self-profile) or at least $R_m \times (1-T)$, where T is a parameter (the *trim* attribute of <*rss*>). The idea is that the "best" Peg in the query should also be reasonably "good" in all the selected samples.

The selected samples are then partitioned into sets indexed by the pair <*lid, lf*>, i.e., two samples belong to the same set, if and only if they come from the same location and are tagged with the same configuration of local features. The idea is that such samples can be treated as a family (different components of the same probe), so, for example, it makes sense to average their readings for a given Peg.

The sets obtained that way are scanned one-by-one and ranked. A “badness” value (the smaller the better) is computed for every set as follows:

1. For every peg *P* of the query, all the readings in the set that include *P* are selected. If the RSS is zero (self-profile), the *self* attribute of <*rss*> (Section 3.3) is consulted. If the attribute is zero (which means that self-profiled values are not acceptable), the reading is ignored. Otherwise, the zero value is replaced with the value of the *self* attribute. If the *plus* flag of the attribute is set, and the RSS value of the query's reading is greater than the value of *self*, the sample's value is replaced by the query's value (effectively resulting in a perfect match). The simple idea is that RSS zero stands for something very high, so it can be assumed to be equal to some fixed

high reading; or, perhaps, any reading higher than some threshold can be assumed to perfectly match the sample's dummy reading.

2. The RSS values (from the query's reading as well as from the sample's readings, after the optional replacement described at point 1) are converted to SLR (an internal signal level representation) based on an interpolation table provided in the data set (element <*rss*>).
3. Let *N* be the number of readings selected in the previous step. Note that we are still looking at a single Peg. The average (*Avg*) and standard deviation (*Std*) are calculated over all converted RSS values (i.e., over the SLR values) from the samples. Note that we mean all readings in the given set that include *P* as the Peg. In a degenerate (but acceptable) case, *N* is 1 (it cannot be zero, because then the Peg from the query would be simply ignored). In such a case, *Std* is set to zero (its value is in fact irrelevant – see below).
4. The badness of Peg *P* is calculated as:

   $$r = \frac{(V - Avg)}{B \times (1-a) + Std \times a}$$

   where

   $$a = \left\{ \frac{N-1}{N} \right\}^S$$

   In the above formulas, *V* is the converted (to SLR) RSS value from the query, and *B* and *S* are parameters (attributes *base* and *shift* of <*classification*>, see Section 3.3). The idea is that the badness (strictly speaking its absolute value) increases with the deviation of the query's reading from the average (the numerator), while being normalized (the denominator) to something that trades between a fixed value (*B*) and the standard deviation observed within the set, in proportion to the number of readings in the set (the more readings, the higher contribution of the standard deviation). When there is a single reading, and no standard deviation can be estimated, the fixed value takes over completely ($a=0$).
5. Once all the Pegs in the query have been examined, it is checked whether *NP* – the number of Pegs that actually have been matched (i.e., subjected to the processing in steps 3 and 4) is not less than NP_{min} (attribute *minimum* of <*classification*>). If *NP* is too small, then the set is skipped and not ranked. An alternative specification of the minimum is as a percentage of the number of Pegs in the query (the *minimum* attribute can provide two numbers – see Section 3.3).

   If the set ends up being ranked, then its total badness is calculated as:

   $$R = \frac{\sqrt{\sum_i r_i^2}}{NP^D}$$

   where *i* ranges over all individual (*NP*) ranks for the Pegs. *D* is a parameter (attribute *length* of <*classification*>) whose role is to weight in (or out) the length of the match (the number of Pegs). For example, it may make sense to assign more importance to longer matches (larger NP) by increasing D).
6. The location attributed to the set with the lowest badness is returned as the location estimate. The algorithm may return the second, third, and so on choice, based on attribute *report* of <*selection*> (Section 3.3).

3 The location server

The server accesses the database read-only and never references the *locations* table. When it returns a location estimate, the location is presented as a number (*lid*). I am not sure if there is a lot of sense in keeping the location names in the database. Custom systems will probably want to keep that data elsewhere.

3.1 Quick start

Execute *make* in the server's directory. The executable will be called *server* (with the extension *.exe* under Cygwin). The server needs the sqlite3 library which should be present in a complete installation of Cygwin.

Having compiled the server, create a database (so the server has something to run on). For that, execute:

```
Scripts/dbase.tcl DBase/dbase.db
```

I am assuming that you are in the main directory of the package. The database maintenance script, *dbase.tcl*, is in *Scripts* and the database will be put into directory *DBase* (file *dbase.db*), where it is expected by the server based on the contents of its XML data set (see *params.xml*).

The script will tell you that the database doesn't exist and will ask if you want it initialized. Having initialized the database, the script will present a prompt:

```
+:
```

indicating that it is willing to accept your commands.

Note: the maintenance script *must* be executed with ActiveState Tcl (*tclsh85*) on my system. This is because the standard version of Tcl available under Cygwin doesn't have the required SQLite extensions. Make sure that the script's header refers to the correct program location of ActiveState Tcl.

Add some data to the database. I have put into subdirectory *SampleData* (looking from the package's main directory) two extracts from files sent me by Wlodek: *loc* (with definitions of some locations) and *samples* (including some samples referencing those locations). The files have been turned into lists of commands to the maintenance script, so you can now enter:

```
file SampleData/loc
```

to enter the location info into the database, followed by:

```
file SampleData/samples
```

to enter the samples. You will see some progress messages. At the end, the script will present its prompt to tell you that it is done. Exit the script by entering:

```
quit
```

The database is now ready, so you may run the server. Execute:

```
./server params.xml
```

The argument tells the server the path to the XML configuration file. Please have a look at the contents of that file. You will see for example that the server is logging its activities into *Logs/logfile.txt*. Inspect that file for any problems. Also, the TCP/IP port of the service is 3445.

Establish a connection with the server. For that execute (from some xterm window):

```
telnet localhost 3345
```

You should see these messages:

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
++olsonet SLOTS location engine version 1.0 ready
```

Now you can enter location queries. For example, try something like this:

```
lo 10 0 100 106 107 120 108 151 104 154 106 125
```

The server will reply:

```
1 loc=1620149 lf=00000000 dist=0.169
2 loc=1590089 lf=00000000 dist=1.324
3 loc=1850142 lf=00000000 dist=2.308
4 loc=2060054 lf=00000000 dist=2.978
++OK
```

The first line is the best ranked guess. The last number in the line gives the badness of the match (see Section 2.2).

Enter:

```
qu
```

to disconnect from the server.

3.2 The protocol

The server accepts just two command types: query and quit. The query format is:

```
lo tag gf peg rss … peg rss
```

where *tag* is the ID of the Tag responsible for the query, *gf* is the set of global features (an integer), and the remaining items provide the readings. All numbers are non-negative. They can be specified as decimal integer or hex numbers with the 0x (or 0X) prefix.

To disconnect, the client can just close the connection. Any of these commands issued by the client:

```
ex
qu
cl
```

will tell the server to gracefully close its end first.

Command keywords are recognized based on the first two letters, so, for example:

```
locate 10 0 100 106 107 120 108 151 104 154 106 125
quit
```

will also work.

A successful location estimate results in a message like the one shown in Section 3.1. Note that such a message always ends with:

```
++OK
```

The number of (ranked) guesses can be up to the limit declared in the XML data file (attribute *report* of <*selection*>) .

A message reporting a failure of the location estimator will start with `--`, e.g.,

```
--no matching samples found
```

You will get the above message, e.g., when there are no samples corresponding to the value of *gf* specified in the query. A non-fatal error message (with the server connection staying alive), begins with !+, e.g.,

```
!+illegal command
```

A fatal message (one after which the connection is dropped) begins with !!, e.g.,

```
!!failed to open the database
```

Note that the parameters read from the XML data file are echoed to the log file (assuming that a log file is set up in the data file), so you can look there to check whether the server has been initialized the right way.

3.3 The parameters

The server accepts a single (optional) argument pointing to the XML data file. Here are the contents of the version of that file that arrives with the package:

```
<parameters>
      <server port="3445">
            <log append="yes">Logs/logfile.txt</log>
            <database>DBase/dbase.db</database>
      </server>
      <engine>
            <rss self="220+" trim="20%">
                  40     0.0
                  240    1.0
            </rss>
            <selection minimum="3" report="4"/>
            <classification
                  minimum="3,60%"
                  base="0.1"
                  shift="1.0"
                  length="0.5"/>
      </engine>
</parameters>
```

The <*server*> section defines three items: the port number of the server's socket, the log file, and the database file. The *append* flag of the <*log*> element indicates that any existing log file with the specified name should be appended to (instead of being overwritten). The defaults are: port 4453, database file *dbase.db*, and no log file.

In the <*engine*> section, the body of the <*rss*> element defines an optional table to be used for converting RSS to SLR (see Section 2.2). The number of values must be even. The first number of every pair stands for an RSS value and the second number gives the corresponding SLR value. The RSS values must be strictly increasing. Note that RSS is always an integer (greater than zero) while SLR can be a floating point number (not necessarily increasing and not necessarily positive). Values in between are interpolated with the first and the last values treated as strict bounds, i.e., any RSS less (larger) than the first (last) value maps into the first (last) SLR.

If there is no <*rss*> element (or if the body of an existing element is empty), no conversion table is defined and the RSS to SLR mapping is an identity function.

The *self* attribute of <*rss*> specifies the RSS value to be assigned to self-profile readings, i.e., the ones with *rss=0* (see Sections 2.1, 2.2). If the value is followed by + (as in the above set), it means that when the reading is compared against a query reading, then any query reading greater than the specified value will be treated as if it were exactly equal the value, i.e., the reading will be considered a perfect match.

There is no default value for *self*. If the attribute is not specified, then the self-profile sample readings (the ones with *rss=0*) will be ignored by the algorithm (as if they didn't exist).

The *trim* attribute of <*rss*> provides the value of *T* used by the algorithm (Section 2.2). It can be specified as a fraction, e.g., 0.2, or a percentage, e.g., 20%. In any case, the value must be nonnegative and less than 1 (or 100%). The default value is 0.2.

The <*selection*> element defines two attributes: *minimum* gives the smallest number of readings in an acceptable query (queries providing fewer readings will be rejected) and *report* tells the maximum number of matches (sorted by their badness) to be shown in response to a (formally valid) location query. The default values are 3 and 1, respectively.

The attributes of <*classification*> translate into these parameters of the algorithm (see Section 2.2): *minimum* → NP_{min}, *base* → B, *shift* → S, *length* → D. The default values are, respectively: 3,50%, 15.0, 1.0, 0.5. Note that *minimum* provides two numbers separated by a comma (the second being optional). The first number gives the absolute minimum for the number of matched Pegs, while the second one (which can be a fraction or a percentage) specifies the fraction of the number of readings in the query. Both limits must be obeyed for a sample set to be ranked.

Note that the XML data file is optional (and there is no default name for that file). When the server is called with no argument, all parameters assume their default values.

4 The maintenance script

The script, named *dbase.tcl*, is invoked with a single argument pointing to the database file. If not specified, the database file name defaults to *dbase.db*. The script starts by trying to open the file and check if it includes the requisite tables (see Section 2.1). If the file doesn't exist, or it doesn't look like a properly initialized SLOTS database, the script offers you an option to initialize it.

Note that at present this is a command-line script; a GUI version may be developed later, if we agree that the idea makes sense. Below is the list of commands. A command is recognized based on the first two letters of the keyword (you can use longer keywords, but anything past the first two letters will be ignored until the first delimiter). The arguments in square brackets are optional. Multiple (exclusive) options are listed separated with |.

`al [-r]` ***`lid name`*** **`[`*`comment`*`]`**

This command adds a new location to the database. If *name* includes blanks, then it can be encapsulated in quotes, e.g.,

```
al 12 "little room next to the loo" used as a storage for junk
```

The above command creates a new location with the ID (*lid*) equal 12. Note that the optional comment can contain any characters: everything following *name* until the end of line is considered a comment.

With *-r*, the command will allow you to overwrite an existing location with the same *lid*.

`sl [-l|-c]` ***`sql_expr`***

This command lists locations. With *-l*, it will only show the *lid* attribute (the short variant), while with *-c*, it will show everything including *comment*. By default, the command only shows *lid* and *name*.

The argument can be any SQL expression (condition) that, in SQL, can be put into a SELECT statement following WHERE that makes sense in the context of the *locations* table (see Section 2.1). For example:

```
sl name glob "room_3*"
```

lists all locations with the *name* attribute starting with the string room_3.

`dl` ***`lid … lid`***

The command deletes the specified locations (identified by the *lid* attribute).

`as` ***`lid tag gf lf peg rss … peg rss`***

The command adds a sample to the database. All arguments are numbers whose meaning was described in Section 2.1. The number of readings (i.e., <*peg, rss*>

pairs) is arbitrary, in particular it can be zero. Readings can be added to a sample later.

Any number can be specified in decimal or in hex (using the 0x or 0X prefix).

The command shows the *sid* attribute of the new sample (assigned automatically by the database) which can be subsequently used as a reference, e.g., when adding readings to the sample.

ar [-r] ***sid peg rss … peg rss***

The command adds a sequence of readings to the sample identified by *sid*. With *-r*, the command will let you overwrite existing readings for the specified Pegs.

sr [-s] ***sql_expr***

This command shows samples along with their complete sets of readings. With *-s*, the sample information (*sid*, *lid*, *tag*, *gf*, *lf*) is only listed once per sample. The argument is an SQL expression that can refer to the tables *samples* and *readings*, e.g.,

```
sr -s (lid < 32 and lid > 30) or peg=109
```

ss ***sql_expr***

This command lists samples (only the attributes kept in the *samples* table). The argument is an SQL expression that can refer to *samples.*

ds ***sid … sid***

The command deletes the specified samples (based on the *sid* attribute) together with their readings.

dr ***sid peg … peg***

This command deletes the specified readings from the sample identified by *sid*.

cl [*min*]

The command cleans the database removing orphaned samples (ones for which there are no locations) and orphaned readings (ones for which there are no samples). The optional argument specifies the minimum number of readings per sample. Any sample with fewer than this many readings will be deleted as well.

fi ***filename***

The command switches the script's input to the specified file. The script will be executing commands from that file until EOF (or *qu*) and then it will switch back to the standard input.

du ***filename***

The database is dumped to the specified file in the form of a list of commands to be executed to restore it. To restore the database you will have to initialize it first and then execute *fi* pointing the script to the dump file.

qu

Quit. The command can be (exceptionally) abbreviated as *q*.

Any unrecognized command (if executed from the standard input) produces the help list presenting all legitimate commands and their arguments.

You cannot use a *fi* command in an external command file. This is to prevent loops.

5 The survey script

The role of this script, named survey.*tcl* (and located in directory Scripts), is to run the database against itself and see how well the samples separate the locations. The script is invoked this way:

```
survey.tcl [-h host] [-p port] [-s samples]
```

with all three arguments being optional. The first two, defaulting to *localhost* and 4453, respectively, point to the location server.[3] The third argument provides the path to the file including the samples. The format of that file should be as that produced by the database dump (Section 4), or of the file *samples* in directory *SampleData*, i.e., it should look as a sequence of '*as*' commands to the maintenance script, with each command defining a complete sample. Any other lines in the file will be ignored. If the file path is not specified (there is no *-s* argument), the script will try to read the samples from the standard input.

The script will issue a query to the server for every sample and analyze the reply. At the end, it will produce (on the standard output) these statistics:

- the total number of lines in the samples file
- the number of samples found in the file
- the number of samples for which the location server has managed to produce an answer
- the number of hits, i.e., samples correctly assigned to their locations
- the number of misses, i.e., mispositioned samples (this number should be equal to the difference between the previous two)

These counters are followed by:

1. The average separation between locations calculated over all samples that were correctly positioned. The separation for one sample is calculated as $S=(B_2-B_1)/(B_2+B_1)$, where B_1 is the badness (*dist*) for the best (correct) estimate, and B_2 is the badness of the first incorrect estimate in the list of estimates returned by the server for the sample.[4] Note that S can take values between 0 and 1 (the higher the better with 1 considered perfect).
2. The minimum separation and the sample for which it was observed.
3. The maximum separation and the sample for which it was observed.

For every mispositioned sample, the script prints a line of text identifying the sample and listing the list of locations returned by the server together with their badness.

For the test database (set up according to Section 3.1), the script should be invoked this way:

```
Scripts/survey.tcl -p 3445 -s SampleData/samples
```

The script will run comfortably under any decent version of Tcl.

[3] Note that the server must be running for the script to work.

[4] Note that the multiple estimates returned by the server (see Sections 2.2 and 3.1) may refer to the same location, but to different sets of local features. If all estimates refer to the same correct location, i.e., ($B_1=B_2$), the separation is assumed to be 1.