IMPLEMENTING THE MOBILE LOCATION PROTOCOL: A TALE FROM THE TRENCHES

Thomas Lindgren Fredrik Linder
Magnus Eklund
Cellpoint AB
e-mail: *first.last*@cellpoint.com

Abstract

We describe the first commercial implementation of the Mobile Location Protocol version 3.0.0. The complexities of finishing the development of a feature-rich protocol in a tight time frame led us to an approach of rapid redevelopment, exemplified below.

INTRODUCTION

Location services are and will remain a distinctive feature of wireless data services in 2G and 3G applications. No dominant standard to access location information has emerged until now: the Location Interoperability Forum, LIF, has recently released version 3.0.0 of the Mobile Location Protocol, MLP. Cellpoint is an active LIF participant, and has developed an implementation of MLP 3.0.0 for use in its mobile location platform. This is the first generally available MLP implementation.

We give an overview of Cellpoint's implementation of MLP and how it fits to the mobile location platform, CMLP, developed by Cellpoint. We conclude with a discussion on development issues.

BACKGROUND

Location services are implemented by interrogating the GSM network on the position of subscribers (or rather, their mobile terminals). This is done by a location server, which provides the service to a collection of clients. A client can be, e.g., a portal deployed by an operator, or a third-party application. Cellpoint has two main products: the mobile location server, MLS, and the mobile location broker, MLB. They are also known as the Cellpoint Mobile Location Platform, or CMLP. The task of the MLS is to provide location information. The MLB acts as an aggregator of location requests and location information by acting as a proxy for multiple MLS:s.

Upon receiving a request, the MLS interrogates the GSM network, using one of several available methods (e.g., cell ID, enhanced cell ID, assisted GPS, or others) [3GPP]. The method used depends on the requested quality of service: some application users require a quick answer, others a precise one. If the subscriber is roaming into another mobile network, the MLS contacts the counterparty MLS, a process known as MLS roaming. For the purpose of implementation, the MLS thus has to act as both client and server.

The MLB has a slightly different role. Each MLB knows of a collection of MLS nodes. When receiving a request, the MLB is responsible for dispatching the request to the right MLS. (This is similar to MLS roaming, but normally occurs inside an operator's network.) The MLB also handles issues such as security, subscriber anonymity, and subscriber privacy, leaving the MLS nodes to act with the MLB as a trusted counterparty. Subscriber anonymity is used to hide the subscriber identity from third parties. Subscriber privacy is used to check whether the subscriber permits the positioning operation.

THE PLATFORM

Each of the CMLP products runs on Solaris and is primarily written in Erlang [2] with supporting code in C and scripting languages. A number of software packages are sourced in to provide the configuration that operators require, e.g., an Oracle database for subscriber information, and Veritas redundant disk software.

When booting, the system first starts an internally developed system for clustering. The clustering system handles failover very quickly (on the order of a few seconds) and provides redundancy for disks [4]. As part of this process, a collection of Erlang nodes are started. When all cluster nodes are operational, the Erlang service software is started as a collection of processes, aka components, that implement the desired functionality. Examples of components are protocol service frontends, subscriber management, database management, the charging subsystem, a WWW-based GUI, the positioning software and the SS7 network interfaces. When all required components are available, the system is ready for service.

THE MLP PROTOCOL

MLP is a complex protocol based on HTTP/1.1 and XML, which has evolved rapidly during the spring of 2002. (As an example, the February 2002 book by Hjelm [6] uses a syntax which is now obsolete.) MLP is being standardized by a committee of operators and equipment providers known as the Location Interoperability Forum, or LIF. At the time of writing, the MLP 3.0.0 standardization process is being finalized [1].

The basic capability is to serve a location request. The client sends a request as follows:

```
<?xml version ="1.0" ?>
<!DOCTYPE svc_init SYSTEM "MLP_SVC_INIT_300.DTD">
<svc_init ver="3.0.0">
  <hdr ver="3.0.0">
    <client>
      <id>application_4</id>
      <pwd>secret</pwd>
    </client>
  </hdr>
  <slir ver="3.0.0">
    <msids>
      <msid type="MSISDN">46702711038</msid>
    </msids>
    <geo_info>
      <CoordinateReferenceSystem>
        <Identifier>
          <code>4326</code>
          <codeSpace>www.epsg.org</codeSpace>
          <edition>6.1</edition>
```

```
      </Identifier>
    </CoordinateReferenceSystem>
  </geo_info>
</slir>
</svc_init>
```

The client sends its identity, password and service ID for authorization. The client can also use a `<subclients>` tag, much as a HTTP proxy, to tell which location servers have participated in the request. Several applications, portals, MLBs and MLSs may be involved in a request, generally for administrative reasons, and must be tracked for correct billing. The subscribers to be positioned are then identified with `<msid>` tags, in this example a single ID of type MSISDN, indicating an ordinary cell phone number, followed by information on the desired quality-of-position (accuracy, timeliness and so on) and the coordinate system to use. Many parameters are optional; in the example, no explicit quality-of-position is given. The coordinate system tells the server how to format the response. Client and server agree on coordinate system by consulting a database maintained by the European Petroleum Survey Group, EPSG, which is the final authority [5].

The server returns a response, e.g., as below.

```
<?xml version ="1.0" ?>
<!DOCTYPE svc_result SYSTEM "MLP_SVC_RESULT_300.DTD">
<svc_result ver="3.0.0">
  <slia ver="3.0.0">
    <pos>
      <msid>46702711038</msid>
      <pd>
        <time utc_off="+0200">20020623134453</time>
        <shape>
          <CircularArea srsName="www.epsg.org#4326">
            <coord>
              <X>20 30 5.4W</X>
              <Y>0 0 3.5N</Y>
            </coord>
            <radius>570</radius>
          </CircularArea>
        </shape>
      </pd>
    </pos>
  </slia>
</svc_result>
```

The response is a position for the indicated MSID, in the shape of a circle centered at the specified coordinate with the given radius. The position was acquired at the time 13:44.53 on 2002-06-23 in time zone +0200.

If there was an error in the request, or the request could not be fulfilled, an error response is returned instead.

This basic functionality is extended by several extra features.

- There is a class of emergency requests with a slightly different format, which are treated with different priority, privacy settings, and so forth by the location server.
- Client-assisted positioning permits the client to send GSM information to another node. This is used to ask a server to map a cell ID (enhanced with, e.g., timing advance information provided by the client) to an actual position.

- Triggering and zoning permits the client to register an interest in a certain subscriber (pending the approval of the subscriber, known as subscriber privacy). When some event occurs, e.g., the subscriber moves into or out of a certain area, the location server contacts the client with this information, inverting the client-server relationship.

THE SIMPL2 PROTOCOL

Before the current product release, Cellpoint used a small, proprietary protocol named SIMPL for positioning requests. Cellpoint decided to use a subset of MLP, extended with some Cellpoint-specific features, as the next version of SIMPL, named SIMPL2.

Cellpoint's SIMPL2 protocol [8] restricts the MLP protocol in several ways, and does not support some of its features. For example, triggering and zoning are not supported, nor are some lesser features concerning format. SIMPL2 reports all such deviations from MLP as errors or unsupported features.

Cellpoint's extensions to MLP concern DES-encryption of requests and responses, and the capability of sending charging information to the charging subsystem of the location server.

DESIGN AND IMPLEMENTATION

The initial specification and implementation proposal for SIMPL2 was written in August-September 2001, based on a preliminary version of MLP 1.1. Business events put the actual implementation on hold until January 2002, with a new development team (the authors) implementing the SIMPL2 version of MLP 3.0.0. The first commercial release of SIMPL2 was in June 2002. Total development time until final delivery was thus roughly 18 man months.

The final development schedule for SIMPL2 was quite aggressive, with prototyping starting quite some time before the standards were even close to finalization. During six months of development (January to June 2002), there were no less than fourteen releases of the SIMPL2 specification, often driven by changes to the underlying MLP being standardized by LIF. Even as late as May, there were major, incompatible changes and extensions in how to specify such things as the coordinate system being used.

At the same time, there was a considerable learning curve for the developers involved: none of us were experienced in the ways of geographical information systems, and the MLP and SIMPL2 standards mainly specified the syntactic formats of requests and responses, rather than the semantics. Thus, we also had to learn how to interpret the requests and translate the intentions to the underlying positioning system.

The CMLP itself furthermore evolved rapidly during this period of time, adding numerous features that required support or changes in the protocol modules. Parts of CMLP were also redesigned to support SIMPL2.

We began by building a basic prototype implementation, integrating the freely available `xmerl` and `inets` subsystems into CMLP and using them for HTTP and XML processing. We then used existing CMLP code for interfacing to the rest of the positioning system, and wrote the code to convert XML to the internal format.

The responsibility of the validation code is to check that the incoming values are valid according to the protocol, and to turn the values into a suitable data structure for the positioning system. A second task is to manage the MLB and MLS roaming functionality, i.e., to act as a client to some other SIMPL2 or MLP server, which requires validation and conversion of responses similar to what was done on requests, and to convert internal data into SIMPL2 positioning requests.

The method of development initially was that of exploratory programming. As the implementation matured, we realized that there were a number of as-yet poorly defined features in MLP and SIMPL2, which, along with changes to the specifications, made necessary continuous fixes to the protocol code.

As an example of the latter, the final draft of SIMPL2 was released on May 12, with seven subsequent drafts released between then and June 7. During the entire development period, there were also extensive mail conversations and revisions in between drafts.

This imposed a considerable burden on us, since Cellpoint also works with a system of weekly formal code freezes that are tested by a separate group of testers. The internal goal was to release a working version of SIMPL2 every week, while tracking changes to specifications, external change requests and protocol bug fixes and clarifications. This proved to be difficult just by patching and debugging the existing code, and we found that our calendars were filled with reacting to such events.

We realized that we needed to retool the code to rapidly react to changes. We decided to implement what we called *rapid redevelopment* to enable us to respond very quickly to changes and trouble reports. This was done as follows.

Abstraction and refactoring

Modularity, macros and abstract data types were used extensively in development. As the implementation matured, we tried to refactor the code further.

We found that Erlangs records were a source of errors. Records are useful because they make changes in data representation easy. The record notation is also handy because it can be used in pattern matching as well as expressions. However, there are also complications:

- There are often constraints between the fields, e.g., when user name has been filled in, then user data fields must be consistent with the name. These constraints are frequently undocumented.
- When new fields are added, every existing record creation in the code must be verified and perhaps updated.

These properties slow down code changes. We opted to throw out explicit use of records and deploy abstract data types instead[1].

As a further benefit, abstract data types also made the code clearer, since the operation is named.

Data driven design

A major source of change in MLP was the format of XML data, both in what attributes were available and their format. Initially, we used a straightforward design to validate formats, structured roughly as follows[2]:

```
check([svc_init, hdr, client, id], ID, State) ->
   check_client_id(ID, State);
```

[1]For readers unfamiliar with this concept, an abstract datatype encapsulates the representation of a data structure inside a module; a well-known Erlang example is the `dict` module. Object oriented programming also uses encapsulation, though with objects instead.

[2]This is not the actual code.

```
check([svc_init, hdr, client, pwd], Pwd, State) ->
   check_client_pwd(Pwd, State);
...
```

This turned out to be a source of problems. The reason was that all aspects of the XML code changed: tag and attribute names, contents, locations in the request, as well as data format names and data format definitions, which quickly became confusing. Not all changes in specification were correctly introduced, and it was difficult to keep track of all the versions.

We refactored validation into an interpreter. The XML attributes and formats were specified as a term:

```
-define(tag_specs,
        [{'svc_init', blank,
         [{'ver', {'or', [{member, ["3.0.0"]}, {unsupported, ver}]}}]},
         ...
        ]).
```

In the case above, the tag `svc_init` has an attribute `ver` which must have the value `"3.0.0"` or the request will get an 'unsupported' error. Furthermore, `svc_init` must not have data content itself.

Each SIMPL2 data type was also defined in a rule engine:

```
apply_rule(blank, Value) ->
    [] == strip_whites(Value);
apply_rule({'or', Rules}, Value) ->
    lists:any(fun(Rule) -> apply_rule(Rule, Value) end, Rules);
...
```

When checking data, the refactored program first locates the rule to use from the specification term, then invokes the correct rule in the rule engine.

We call this a *data driven design*. The net effect is to separate the *adminstrative* aspects from the *specificative* aspects of checking.

Ensuring that we followed the current specification was simplified immensely, since the specification part changed most rapidly, and this was now much easier to verify quickly. In the first part of the project, testers found most of the data format bugs. As we neared delivery, the rules engine approach had overcome this problem and we could instead turn to fine-tuning the definitions in the SIMPL2 specification.

Testing framework

As we have described, SIMPL2 is a very flexible protocol with many features to implement and test.

To improve internal quality assurance at releases, we implemented a testing framework, also in Erlang. We decided to implement an external testing framework, but not one for unit testing. The reason was that automated unit testing seemed harder to get right and Erlang's purity and natural interactive bottom-up development at the same time made reasonable manual unit QA more straightforward than one might expect.

Our testing framework generates requests and the expected responses, sequentially tests every case, and signals any deviant responses. At present, there are roughly a hundred test cases. We also implemented a regression testing framework to ensure that old bugs did not reappear: when there is a bug report, we write a test case that triggers the bug and re-run the test at subsequent releases.

The testing framework is data driven: each test request is specified as an Erlang data structure, which is translated into XML text. Specifying a series of test requests looks like:

```
test_series(1, 1) ->
   Clients = [{"service_a", "secret", ?OK}, ...],
   MSID = "...",
   [ {Expect,
      ?svc_init(?hdr_client(Name, Pwd), ?slir(?msids(MSID),
                ?default_geo_info))
     || {Name, Pwd, Expect} <- Clients ];
...
```

This generates one test case per client, with an expected outcome Expect and a request to be sent. The list of Clients consists of cases that should pass or be rejected, as specified by the programmer.

The testing framework sends each request to the server under test, awaits the response and notifies the user when there is an unexpected response (ranging from socket errors to XML validation errors).

Social aspects

Our development methods have also made extensive use of pair programming, brief coordination meetings (sometimes several per day) and the use of instant messaging to keep developers synchronized, even though there were only three main developers (along with half a dozen developers working with related items and three testers). This has improved code quality by keeping all developers "working in the same paradigm". The discussions have also turned out to help with finding and resolving the grey areas of SIMPL2 and MLP; at least one of our findings has led to changes in the MLP standard specification.

DISCUSSION

How did rapid redevelopment work out in practice? Quite well.

Since the XML tags, attributes and validation rules changed very frequently, the rules-based approach to validation was extraordinarily helpful: the rules are easy to read, understand and change compared to code. Verifying that the implementation conforms to the specification is much easier.

A second improvement is that using abstract data types meant we could control how data was manipulated, we could validate inputs and outputs and we could change representations easily. By not exposing record datatypes, we ensure discipline in how crucial data is created, used and accessed[3].

Because of the many features of SIMPL2 and their potential interactions, our testing framework has been invaluable. As a trouble report appears and is fixed,

[3]This requires programmers not to break data abstractions, rather than prohibiting them from doing so as do, say, SML:s abstypes.

we add a regression test case. Prior to each weekly release, we check that the code passes all the tests. We then hand it off to the testers for formal quality assurance.

To improve on this, it would probably be useful to have a dedicated tester to add more test cases. While doing this full time for just SIMPL2 would be overkill, it might be a very useful part-time effort.

One can view rapid redevelopment as a form of extreme programming [3]. Here is a comparison. The item bullets are taken from the reference, page 54.

Small releases. We incrementally released code every week once the system was up and running. However, the QA organization had problems in tracking what could be usefully tested. Serious formal testing began only in the later parts of the project.

Testing. XP mandates a strict testing regime. We did not use XP unit tests, but implemented and used an automated testing framework successfully, as described previously. The effect was gratifying, since it improved code quality while reducing our testing effort.

Refactoring. We refactored code thoroughly, as shown previously, but mainly in reaction to problems rather than proactively. This might have been a mistake.

Pair programming. Considerable portions of the code were developed using pair programming, which turned out to be a good choice.

Continuous integration. Erlang naturally enables continuous integration. The normal development cycle is to edit-debug-recompile, then load the code into a running full CMLP system (configured as an MLS or an MLB) to test it.

However, we also had to integrate the results of other developers. This was done at every release and included changes in the system configuration files (including their format) and CMLP restarts. Continuously integrating such changes would likely have slowed development down.

On-site customer. We did not have an on-site customer. As mentioned above, a dedicated test case writer would be useful. Also, having a customer representative available would have clarified a number of practical issues. The main issue in introducing this is probably to motivate the expense of doing it.

Coding standards. We did not use written coding standards, but some principles were obeyed, such as the use of ADTs instead of records.

FUTURE WORK

We are currently mulling over if the code that converts incoming data to our internal format, and back into outgoing data, can be converted into a rule-based form as well. The main problem is that our current set of sketched conversion rules is too large and unstructured to yield any great advantage on ordinary code.

Given the successful rules-based approach to validation, we are also considering writing a tool to translate the rules into a directly validating and translating XML parser. The advantage of this approach is higher efficiency (as long as code size remains reasonable).

However, it is not at all clear that SIMPL2 is a bottleneck. Positioning requests may naturally have long latency (e.g., several seconds), and the cost for SIMPL2 processing is small in this context. While reducing latency and memory footprint per request is still welcome, because system capacity is improved, other tasks have had priority since SIMPL2 was released to customers.

CONCLUSION

We have described the implementation of the SIMPL2 version of MLP 3.0.0 on Cellpoint's location server series, CMLP.

SIMPL2 evolved quickly and had many features; in this context, rapid redevelopment, an XP-like approach based on abstraction and testing, has been highly successful.

In a development project concerned with a large and changing feature set and an aggressive schedule, the resulting emphasis on flexibility and ease of modification has been invaluable in reducing trouble response times. This in turn leads to a virtuous spiral, where development gets the time to rewrite and extend the code before testing, rather than as a response to testing finding bugs.

We have thus found rapid redevelopment to be a fruitful way to develop feature-rich, committee-specified protocols with high demands on reliability, as is often the case in the telecom world.

A version of CMLP including SIMPL2 is commercially deployed at an operator at the time of writing, making this the world's first implementation of the MLP 3.0.0 standard.

KISTA, JUNE-OCTOBER 2002

ACKNOWLEDGEMENTS

The comments of Lars-Göran Ericson and Bogumil Hausman were helpful in preparing this paper.

References

[1] 3GPP, standard document 03.71. http://www.3gpp.org

[2] Joe Armstrong, Robert Virding, Claes Wikstrm, Mike Williams. Concurrent Programming in Erlang. Prentice Hall.

[3] Kent Beck. Extreme Programming Explained. Addison-Wesley, 2000.

[4] Per Bergqvist. Improving Robustness in Distributed Systems. Proc. Erlang User Conference, 2001.

[5] European Petroleum Survey Group. http://www.epsg.org

[6] Johan Hjelm. Creating Location Services for the Wireless Web. Wiley, February 2002.

[7] Location Interoperability Forum. LIF Mobile Location Protocol. LIF TS 101 v3.0.0, rev 2. June 3, 2002.

[8] Rob Schmersel. SIMPLv2.0 specification. Internal Cellpoint document.