

Design and implementation of the middleware of Sputnik

Hannes Mehnert
hannes@mehnert.org
Dylan Hackers

ABSTRACT

This paper describes the design and implementation of the middleware of Sputnik, an RFID tracking system with distributed sensors and 3D visualization. The middleware of Sputnik consists of a communication layer, some basic data mining, a persistent store and a HTTP interface. The time frame was 8 weeks part-time for the middleware. Scalability was a requirement. The author used Dylan, a compiled, object-oriented, dynamic typed programming language. The paper demonstrates that Dylan is capable of both scalability and fast development. The paper gives an overview of the RFID tracking system and of Dylan. Then the main part describes the design and implementation of the middleware. This is followed by a conclusion and planned improvements for the next buildup.

1. SPUTNIK - AN RFID TRACKING SYSTEM

The paper consists of design and implementation of the middleware of an RFID tracking system with distributed sensors and 3D visualization. This section gives a more detailed overview of the RFID tracking system and the involved teams. Section 2 describes goals of the middleware. In section 3 the author introduces Dylan and rationalizes why he chose Dylan for this project. Section 4 describes design and implementation of the middleware. In section 5 the author gives a conclusion and introduces further development.

The purpose of Sputnik was to make attendees of a conference (23rd Chaos Communication Congress) aware of privacy issues involved in mobile, wireless technologies like RFID. Sputnik is a combination of custom RFID hardware and a 3D visualization. The hardware consists of RFID readers and wearable tags. The 3D visualisation consists of a 3D renderer, a 3D model of the building and a virtual attendee for each tag rendered in real-time. The middleware is the glue between both parts. Table 1 shows the involved

teams.

1.1 Hardware

The RFID team designed hardware and software of a custom RFID reader and RFID tags for this project. Hardware layout and software of the tags are open and freely available. The RFID readers are a microcontroller with a small firmware written in C. At the conference the RFID team installed about two dozen RFID readers in the building. Each RFID reader transmits RF (radio frequency) fields in a short time interval. The RFID tags answer to these fields with the transmitted field intensity, an ascending sequence number and the tag identifier. The location of the RFID reader is used as a simple approximation for the position of the tag. The RFID readers convert the answers from the tags into binary UDP packets.

1.2 Middleware

The middleware consists of a communication layer between the RFID readers and 3D visualization, as well as a persistent store which stored information about tags, users and their associations. Furthermore it contains location tracking and data mining of the location profiles. Additionally a HTTP interface which displays information about users and their locations, allowing to change those. The author chose Dylan as implementation language.

1.3 Visualization

The 3D visualization is a 3D rendering engine and a custom plugin which is capable of displaying avatars. Its communication interface is XML messages via HTTP. Because tag positions are only approximations, avatars are positioned randomly around the position of the observing reader. The user interface is a touch-screen. A screenshot is shown in figure 1. On the left information about the selected avatar is displayed. If a user is associated with the tag identifier, additional user information is also shown. On the right floor plans of all levels of the building are displayed. Each red point represents a virtual observation camera. The main part is the point of view of the selected virtual observation camera. This could be rotated with a knob.

2. DESIGN GOALS OF THE MIDDLEWARE

The time frame for the project was short, eight weeks from the first meeting until the conference where it should be set



Figure 1: Screenshot of the 3D user interface

Task	Team	URL
3D visualization	ART+COM	http://www.artcom.de/
3D model	Marplon4	http://marplon4.de/
Coordination	Tim Pritlove	http://tim.pritlove.org/
Middleware	Hannes Mehnert	http://sputnik.dylan-user.org/
RFID Hardware	OpenPCD	http://www.openbeacon.org/

Table 1: Project teams

up. The author volunteered to do this in his part time. Requirements evolved and changed during development by the other teams. The middleware should be capable of processing packets from 1000 tags easily, demanding high performance.

2.1 Communication

There are multiple design goals of the middleware, the most important is communication between the hardware and visualization team. The hardware team send UDP packets for each observed tag, and the visualization team needs observations as XML messages.

The UDP packets have no position information of the tag. The position information has to be derived from the position of the observing RFID reader. The visualization demanded a position information in the XML observations.

There are several algorithms for location singularization, so an interface to easily change an algorithm or write a new one is demanded. This resulted in a requirement of a XML multiplexer via TCP, everyone should be able to connect, receive and transmit observations.

Figure 2 shows the data flow of Sputnik. On the left side an RFID tag carried by an attendee transmits a radio signal which is received by the access point. The signal is encrypted with XXTEA. The access point converts this to binary packets and send them to the decrypter via UDP. It decrypts the packet and transmits it via UDP to the UDP collector.

The UDP collector transforms the packets to XML observations and adds a received timestamp. Those XML observations are send via TCP to the XML multiplexer, called reflector. It listens for a HTTP request and multiplexes all received XML observations. The visualization interface is a client of the reflector and receives XML observations from the reflector. The data mining modules are also connect to the reflector. They analyze the received XML observations and send back more precise information about tag locations.

2.2 Data store and HTTP interface

A wearer of a tag should be able to add additional data to the tag, nickname, gender, mugshot, etc. This information is used in the visualization by displaying different avatars, black are private avatars with no additional information, white are avatars which have additional information attached. This can be displayed by selecting the avatar. The additional data has to be available via HTML via HTTP, serving as a user interface, as well as via XML via HTTP to be processed by the visualization automatically. Obviously, the data has to be stored persistently.

3. DYLAN

3.1 Language Overview

Dylan was developed about 15 years ago by CMU, Apple and Harlequin. It provides several abstractions known from Lisp, but has a clear separation of compile time and run time, thus providing more performance and a smaller memory footprint, the compiler is not in memory when the program is executed.

A short overview of Dylan is given in the citation:

'Dylan is an advanced, object-oriented, dynamic language which supports rapid program development. When needed, programs can be optimized for more efficient execution by supplying more type information to the compiler. Nearly all entities in Dylan (including functions, classes, and basic data types such as integers) are first class objects. Additionally Dylan supports multiple inheritance, polymorphism, multiple dispatch, keyword arguments, object introspection, macros, and many other advanced features...' – Peter Hinely <http://www.opendylan.org/>

A more detailed overview of the features is given in the remainder of this subsection.

Dylan is an object-oriented programming language [drm]. It also has functional aspects, higher order functions and anonymous functions. Everything in Dylan is an object, inheriting from the class `<object>`. The object system is similar to the Common Lisp Object System. Type annotations are optional, and default to `<object>`. The design of Dylan was mainly influenced by Lisp and Smalltalk. Its syntax is an Algol like prefix syntax. Dylan provides automatic memory management.

Dylan is strongly typed, hence a developer is not able to work around the type system with pointer arithmetic, providing safety. Dylan is also dynamically typed, thus runtime types are relevant for method dispatch.

A major difference between Dylan and other object-oriented programming languages is the separation of classes and functions. Functions and classes are defined in the first-class namespace. Classes reflect data structures, while functions are algorithms working on instances of classes. This provides a clear separation of data structures and algorithms.

A class consists of a name and the superclasses. By convention class names are surrounded with angle brackets like `<object>`. Dylan provides multiple inheritance by doing superclass linearization. This prevents the problem that a member variable can be inherited multiple times as it is in C++. A class contains a list of slots, called members in other programming languages. Access to a slot is wrapped by getter and setter methods. Also, syntactic sugar allows to specify default init values, init keywords, etc. during class definition.

A class definition of a `<square>` can be

```
define class <square> (<object>)
  slot height :: <integer>, init-keyword: height;;
  slot width :: <integer> = 0, init-keyword: width;;
end
```

This defines the class `<square>`, which contains two slots, a width and a height. Both are of type `<integer>`, and can be set during instantiation by providing the keyword height or width. Width is initialized to 0 if no keyword is specified.

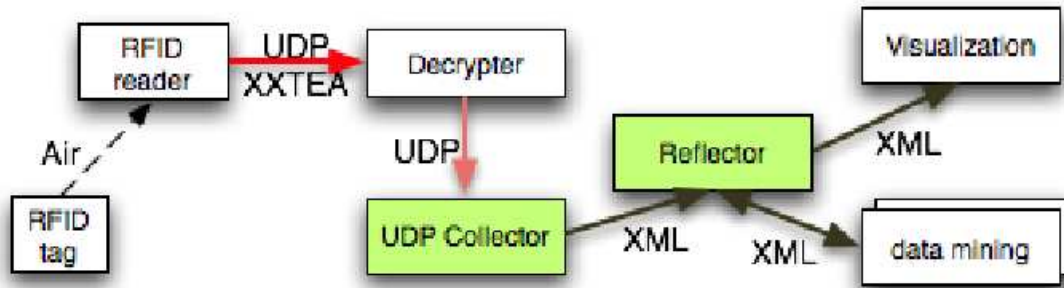


Figure 2: Data flow of the communication layer

Another class definition can be a <circle>

```

define class <circle> (<object>)
  slot radius :: <integer>, init-keyword: radius;;
end
  
```

A generic function has a name and a list of required arguments. Methods are added to the function by providing more specific types of the arguments. When a generic function is called, the method which has the most specific types of all arguments is called. This concept is known as multiple dispatch.

A sample usage of a generic function is to calculate the area of a geometric object:

```

define method area (s :: <square>)
  => (res :: <integer>)
  s.width * s.height
end;

define method area (c :: <circle>)
  => (res :: <integer>)
  c.radius * c.radius * \ $pi
end
  
```

Since area only has one argument, single dispatch is done. Depending on the type of the object, either the first or the second method is used to calculate the area of the geometric object.

To improve performance, and eliminate generic function dispatch at compile time, types can be annotated. Additional performance can be achieved with a concept called sealing. This allows a developer to define that a method can not be further specified outside the current compilation context. Also classes can be prohibited from getting subclassed outside of the compilation context. These lead to a less dynamic system, but provides several optimization possibilities.

A compilation context is a library, consisting of one or more modules. Each library can import other libraries and export modules. Each module can import other modules and export bindings, which are variables, constants, methods,

classes or macros. By exporting only the getter or both getter and setter or neither of a class slot, the access is protected, public or private.

Dylan also provides a meta-programming facility similar to Lisp macros, but less powerful. The macro system provides the possibility to do pattern matching and substitution on fragment types. This is in lots of cases powerful enough.

Two open source Dylan compilers are available. Both are written in Dylan. `d2c` was developed at CMU and translates Dylan code to C. `opendylan` was developed at Harlequin and translates Dylan code to x86 assembler or C. Because `d2c` still lacks thread support, the author used `opendylan` for this project. `opendylan` is a complete IDE, a profiler, hot code update and an integrated debugger. This currently only works on Windows. A batch compiler is available for Linux, FreeBSD and MacOSX.

3.2 Statement why Dylan was used

The author chose Dylan because he was already familiar with Dylan. The short time frame demanded a programming language where a developer does not have to care about low level aspects like memory management. A scripting language like Ruby or Python would have been not a wise choice, since lots of performance was demanded.

C would also be not a suitable choice because everything has to be taken care manually. The amount of errors easily done in C, like off-by-one in loops or memory management, as well as pointer dereferencing, is too high for a project which has to be finished in a short time frame.

Also the amount of abstractions provided, namely several syntactic sugar, a class hierarchy with multiple inheritance, multiple dispatch, as well as a convenient module system and meta-programming system, prevent a developer from the need to write boilerplate code and code duplication is minimized.

4. SOLUTION FOR THE MIDDLEWARE

4.1 Communication layer

4.1.1 UDP collector

The UDP collector listens for UDP packets from the RFID access points and transforms them to XML observations.

These are sent to the visualization.

The UDP packets contained the following information: version number, tag identifier, sequence number and receive-power. Parsing of binary data can be done with the Packetizer framework [ilc07] in Dylan. It is a domain specific language (DSL) using the Dylan macro facility to define a binary packet in a concise way. To parse and assemble the binary UDP packets, only the following protocol definition is needed.

```
define protocol sputnik-udp-frame (container-frame)
  summary "from %s %s tx %d ID %= S %=",
    originator, flags-summary, transmit-strength,
    unique-tag-id, sequence-number;
  over <udp-frame> 2342;
  field originator :: <ipv4-address>;
  field data-size :: <unsigned-byte>,
    fixup: byte-offset(frame-size(frame));
  field protocol-version :: <unsigned-byte> = 23;
  field reserved :: <6bit-unsigned-integer> = 0;
  field flag-sensor :: <1bit-unsigned-integer> = 0;
  field flag-ack :: <1bit-unsigned-integer> = 0;
  field transmit-strength :: <unsigned-byte>;
  field sequence-number
    :: <big-endian-unsigned-integer-4byte>;
  field unique-tag-id
    :: <big-endian-unsigned-integer-4byte>;
end;
```

The Packetizer translates the protocol definition to a class definition, providing accessors to the fields which instantiate Dylan objects as lazy as possible. There are three different methods available, `parse-frame` (`frame-type :: subclass(<frame>)`, `byte-vector :: <byte-vector>`) \Rightarrow (`frame :: <frame>`), `assemble-frame` (`frame :: <frame>`) \Rightarrow (`byte-vector :: <byte-vector>`) and `summary` (`frame :: <frame>`) \Rightarrow (`string :: <string>`).

The visualization team specified with the author the following XML schema definition for observations:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name='observation'
    <xs:attribute name='observer' type='xs:anyURI'
      use='required' />
    <xs:attribute name='observed-object' type='xs:anyURI'
      use='required' />
    <xs:attribute name='time' type='xs:double'
      use='required' />
    <xs:attribute name='priority' type='xs:integer'
      use='required' />
    <xs:attribute name='min-distance' type='xs:float'
      default='0' />
    <xs:attribute name='max-distance' type='xs:float'
      default='255' />
    <xs:attribute name='direction' type='xs:string'
      default=' [0,0,0]' />
    <xs:attribute name='message' type='xs:string' />
    <xs:attribute name='tags' type='xs:string' />
    <xs:attribute name='position' type='xs:string' />
```

```
</xs:element>
</xs:schema>
```

The time attribute is a double float containing seconds.microseconds since 1970-01-01. The direction and position attribute have the same syntax, a vector of floats ('[x,y,z]').

A Dylan class for `<observation>` looks similar to the XML schema definition:

```
define open class <observation> (<object>)
  slot observer :: <symbol>, init-keyword: observer;;
  slot observed-object :: <integer>,
    required-init-keyword: object;;
  slot timestamp :: <date> = current-date(),
    init-keyword: time;;
  slot min-distance :: <float> = 0.0s0,
    init-keyword: min-distance;;
  slot max-distance :: <float> = 255.0s0,
    init-keyword: max-distance;;
  slot direction :: <string> = "[0,0,0]",
    init-keyword: direction;;
  slot priority :: <integer>, init-keyword: priority;;
  slot message :: false-or(<string>) = #f,
    init-keyword: message;;
  slot tags :: false-or(<string>) = #f,
    init-keyword: tags;;
  slot position :: false-or(<position>) = #f,
    init-keyword: position;;
end;
```

Dylan has a XML parser library, which can parse XML from a stream, and translates XML observations to instances of `<observation>`.

To transform an `<observation>` to a XML string representation, the method `encode-xml` is defined. It uses the macro `with-xml` which produces a XML element. This method returns an `<element>` which can be converted to a `<string>`.

```
define method encode-xml (obs :: <observation>)
  => (res :: <element>)
  let obs-obj = integer-to-string(obs.observed-object);
  let observation =
    with-xml ()
      observation(observer =>
        concatenate($cps-url,
          "tracking-observers/",
          as(<string>, obs.observer)),
        id => obs-obj,
        observed-object =>
          concatenate($cps-url, "tracking-users/",
            obs-obj),
        time => get-time-as-string(obs.timestamp),
        priority => integer-to-string(obs.priority),
        min-distance =>
          print-float(obs.min-distance),
        max-distance =>
          print-float(obs.max-distance),
        direction => obs.direction)
```

```

end;
if (obs.position)
  let att =
    make(<attribute>,
        name: "position",
        value: as(<string>, obs.position));
    add-attribute(observation, att)
  end;
  observation;
end;

```

A `<raw-observation>` is a subclass of `<observation>` with a priority of 0.

```

define class <raw-observation> (<observation>)
  inherited slot observation-priority :: <integer> = 0;
end;

```

To convert from UDP binary packets to XML and the other way around, only a few lines are needed, as shown before: class definitions of the binary protocol, of an observation, and methods converting from and to XML string representations.

To actually receive UDP packets and transmit them as XML, a singleton class `<udp-collector>` is defined. It stores the latest sequence number of each tag and pushes each observation to an output stream. The class definition of a `<udp-collector>`:

```

define class <udp-collector> (<single-push-input-node>)
  constant slot sequence-number-table :: <table>
    = make(<table>);
  constant slot output-stream :: <stream>,
    required-init-keyword: output-stream;;
end;

```

It inherits from `<single-push-input-node>` which is defined in the packet flow graph library of the TCP/IP stack written in Dylan [dylantcpip]. A `<single-push-input-node>` obviously has one push input. This is connected to a demultiplexer output which filters UDP packets to the specific destination port. The method `push-data` is called for each packet.

```

define method push-data
  (node :: <udp-collector>,
   frame :: <sputnik-udp-frame>)
  let last-sequence
    = element(node.sequence-number-table,
              frame.unique-tag-id,
              default: 0);
  if (last-sequence < frame.sequence-number)
    node.sequence-number-table[frame.unique-tag-id]
      := frame.sequence-number;
  let source = frame.originator;
  let obs =
    make(<raw-observation>,
        observer: as(<symbol>,
                    as(<string>, source)),

```

```

        object: frame.unique-tag-id,
        max-distance:
          frame.transmit-strength / 5.0s0);
    write(node.output-stream, encode-xml(obs));
  end;
end;

```

It checks whether the sequence number is larger than the previous one. If that is the case, it updates the sequence number for the tag in the `sequence-number-table` slot. It also instantiates a `<raw-observation>` and writes this on the output stream. The output stream was `*standard-output*` for debugging, a TCP stream for production use.

Initialization code which opens the socket and output stream as well as connects the `<udp-collector>` object in the flow graph is omitted for brevity.

The author implemented a native interface to pass observations around to improve performance. Before every Dylan library parsed a XML observation and instantiated an `<observation>`. The native interface required a new slot in `<udp-collector>` named `push-closure` which is by default initialized to `compose(curry(write, node.the-output), encode-xml)`. The `<udp-collector>` uses `node.push-closure` instead of `write` to send data.

Translating raw UDP packets to XML observations is easily done with Dylan, using the Packetizer DSL and a few lines of code doing the actual work. It would be useful to have an automated way to convert XML schema definitions to Dylan classes, providing transformers between XML string representation and Dylan objects.

4.1.2 XML multiplexer

A XML multiplexer which receives XML observations and transmits them to multiple clients via TCP is called reflector. It opens a TCP listener socket and waits for incoming connections. Each time the reflector accepts a connection, it creates client thread. This thread parses the HTTP request line and creates a `<reflector-client>` object with the parsed priority. The client thread adds this object to the list of clients of the singleton `<reflector>` instance. Afterwards the client thread waits for XML observations on the stream. The client gets a notification every time a XML observation with the specific priority is multiplexed and needs to be sent out. To prevent loops, clients have to send observations with a higher priority than the requested priority. A special priority of 0 specifies raw observations, those are observations which have only converted from UDP, but contain no additional information.

A `<reflector>` has separate lists, one for raw-clients and a sorted one for non-raw clients. The reflector also has a cache for the mapping of observation priority to index into the client list, because a new observation occurs more often than a client entering or leaving. This mapping is stored in the `<reflector>` in a table and trashed when a client is added or removed. It is filled when an observation with a priority not in the cache is multiplexed.

As already mentioned, a major performance im-

provement was to pass Dylan objects to functions instead of XML observations over TCP streams. The author enhanced the `<reflector>` class with a list and cache of `<reflector-dylan-client>`. A `<reflector-dylan-client>` is a subclass of the abstract `<reflector-client>` which contains a single slot `priority :: <integer>`. A `<reflector-dylan-client>` has an additional slot `closure :: <function>`. The reflector calls the closure instead of `write`. Another subclass of `<reflector-client>` is `<reflector-tcp-client>`, which provides the functionality of the previous `<reflector-client>`. A bidirectional communication between TCP clients and native Dylan clients is implemented.

The reflector reuses the observation class from the UDP collector to parse a XML observation and retrieve the priority. It is a multi-threaded program where each client thread has a notification and a queue in which observations are stored by the thread which received the observation.

4.2 Data mining

To implement virtual sources which add data to observations like position information or which tag attended which talks, an interface is specified. This contains the class `<virtual-source>` which has a slot `priority :: <integer>` specifying the priority of observations it receives. The virtual source also has a slot `push-closure :: <function>` which it calls each time it generates a new observation. The method `receive-data (source :: <virtual-source>, observation :: <observation>)` is called by the reflector. The method can be extended for subclasses of `<virtual-source>`. Each `<virtual-source>` subclass instantiates a different `<observation>` subclass, which contains the priority of the observation.

4.2.1 Location tracking

The 3D visualization needs location information to display avatars. Raw observations do not contain location information, leading to the development of a location tracking library. This library gathers information from two sources, the observers IP address of each observation and a mapping from IP address to relative position. The latter is provided in a XML file.

```
define class <dumb-virtual-source> (<virtual-source>)
  inherited slot priority = 0; //listen for raw data
  slot observers :: <table> = make(<table>);
end;
```

The `<dumb-virtual-source>` receives raw observations (priority 0). A mapping from the IP address of the observer to location is stored in the slot `observers :: <table>`. This table is filled by reading a XML file on startup and the `xml-element-handler` shown next.

```
define method xml-element-handler
(symbol == #"observer",
 message :: <element>,
 source :: <dumb-virtual-source>)
```

```
  let fa = curry(find-attribute, message);
  source.observers[fa(#"observer-id")]
  := as(<position>, fa(#"position"));
end;
```

The streaming XML parser calls the generic function `xml-element-handler` with a XML tag name, a XML element and an object. The displayed method is specified on `#'observer`, a XML `<element>` and `<dumb-virtual-source>`. It adds an entry to the observers table, using the observer-id as key and the converted position as value.

```
define class <dumb-observation> (<observation>)
  inherited slot priority = 23;
end;
```

A `<dumb-virtual-source>` produces `<dumb-observation>` objects, which have a priority of 23. They are received and used in the 3D visualization.

```
define method receive-data (s :: <dumb-virtual-source>, o :: <observation>)
  let dumb-observation
    = make(<dumb-observation>,
          object: o.observed-object);
  if (element(s.observers, o.observer, default: #f))
    dumb-observation.location := s.observers[o.observer];
  s.push-closure(dumb-observation);
end;
end;
```

The reflector calls `receive-data` for each raw observation. `receive-data` generates a `<dumb-observation>`. If the position of the RFID reader is found, the location slot of the `<dumb-observation>` is set to this position. The observation is then sent via the `push-closure`.

Because of the one-to-one mapping of a `<raw-observation>` to a `<dumb-observation>`, too many observations were produced. To decrease the volume each incoming observation was stored in a table indexed by the observed-object destructively. Every 5 seconds a worker thread emits an observation for each tag in the table. Because all observations but the last in each time interval were not processed this was neither a good solution.

A further improvement was to store the observations of each tag in a list. Every 5 seconds the average of all observed positions weighted by signal strength were computed.

4.2.2 Lecture matching

A first data mining service is storing which tag attends which talks. A XML file provides information when and where lectures are hold. If a tag is in a room for 5 minutes and a lecture is in this room, the specific lecture is added to the list of attended lectures of the tag. A mapping between positions and rooms is needed for this, but was not finished yet.

A `<room>` basically has a name and a list of `<cuboid>`s.

```

define class <room> (<object>)
  constant slot room-name :: <string>,
    required-init-keyword: name;;
  slot cuboids :: <collection> = make(<stretchy-vector>)
end;

```

Each cuboid contains of two positions, the front lower left and back upper right point.

```

define class <cuboid> (<object>)
  constant slot lower :: <position>,
    required-init-keyword: lower;;
  constant slot upper :: <position>,
    required-init-keyword: upper;;
end;

```

The author implemented a function `tag-in-room?`. This was trivial with higher order functions: `any?(curry(tag-in-cuboid?, tag.position), room.cuboids). tag-in-cuboid?(position :: <position>, cuboid :: <cuboid>)` checked whether position is in cuboid with `(position < cuboid.upper) & (cuboid.lower < position)`. `<(a :: <position>, b :: <position>)` checked whether all three dimensions of a are smaller than b, `(a.x < b.x) & (a.y < b.y) & (a.z < b.z)`.

4.3 HTTP interface

A HTTP interface served as a pull interface for user information, as well as user registration interface and storage of associations between tags and users.

Koala, a web server written in Dylan, was used for this task. It provides dynamic pages, dylan server pages (DSP), which are similar to java server pages. A developer defines tags in Dylan code to use in DSP template files.

For persistent storage DOOD, an object store, is used. It can serialize and deserialize any Dylan object to and from a stream. A global table containing all persistent objects is kept in memory and dumped periodically onto disk.

The HTTP interface provides access to four different types of data: users, tags, observers and rooms. A user has a name, a list of associated tags plus some additional information. A tag has a list of observations and an associated user optionally. The purpose of storing observers and rooms was to have a central configuration storage.

The HTTP interface returns XML elements of requested observers, rooms and tags with `/tracking-<class>/id0/.../idX`. The 3D visualization uses these requests to collect more information on a specific object.

The HTTP interface also renders HTML pages, providing four different operations on the stored data types: adding, editing and viewing of an instance as well as viewing all objects of the specific type.

The main DSP handler a page context is passed around. This is a subclass of `<dylan-server-page>`. There are two

orthogonal properties of a page, operation and object type. Multiple inheritance is used to inherit from both the specific operation class and the type of the object. If all combinations are possible this would result in 16 classes.

Writing 16 classes by hand is error-prone. That is why the macro `sputnik-page-definer` is used.

```

define macro sputnik-page-definer
  { define sputnik-page ?name () ?storage:expression end }
=> {
  define abstract class "<" ## ?name ## ">" (<sputnik-page>)
    inherited slot page-title = ?"name";
  end;
  define page ?name ## "-detail-view"
    ("<" ## ?name ## ">", <detail-view-sputnik-page>)
    (url: "/" ## ?"name" ## "-detail",
     source: ?"name" ## "-detail.dsp")
  end;
  define page ?name ## "-list-view"
    ("<" ## ?name ## ">", <list-view-sputnik-page>)
    (url: "/" ## ?"name" ## "s",
     source: ?"name" ## "-list.dsp")
  end;
  define page ?name ## "-edit"
    ("<" ## ?name ## ">", <edit-sputnik-page>)
    (url: "/" ## ?"name" ## "-edit",
     source: ?"name" ## "-edit.dsp")
  end;
  define page ?name ## "-add"
    ("<" ## ?name ## ">", <add-sputnik-page>)
    (url: "/" ## ?"name" ## "-add",
     source: ?"name" ## "-add.dsp")
  end;
  define method get-objects-of-page
    (page :: "<" ## ?name ## ">") => (res :: <table>)
    storage(?storage);
  end; }
end;

```

Macros work on fragments in Dylan, macro variables like `?name` on the left hand side are available by `?name` on the right hand side. `##` is the splicing operation.

The macro expands `define sputnik-page user () <sputnik-user> end` to an abstract superclass `<user>`. It also generates a subclass for each operation which defines the URL it responds to and the DSP file it uses. Additionally the macro provides the method `get-objects-of-page` which returns all objects of the page type.

The URL schema is `/<class-name>-<operation>/<object-id>` where object-id is only relevant for edit and detailed view pages.

Not everyone should be able to add and modify everything. The author implemented authentication via HTTP [RFC 2617]. Permissions are defined as following:

- adding a `<sputnik-user>` could be done by everyone
- editing a `<sputnik-user>` could only be done by himself

- admins were allowed to add and edit everything

The initial responder for all URLs first checks whether a HTTP authenticate header is supplied. If so and authentication is successful, it binds the thread-local variable `*user*` to the user-object extracted from the list of users. If no authenticate header is supplied and the page is public viewable, it is shown. If it is an add or edit page, a WWW-authenticate error is sent back.

The next method is dispatched on the `<sputnik-page>` subclass.

A handler for `<list-view-sputnik-page>` binds the thread variable `*object*` to the list of objects of the specified type and processes the DSP file which iterates over all elements of the list and displays a summary information of each object.

The handler for `<add-sputnik-page>` processes the DSP after checking whether `*user*` has admin privileges. A special case is `<user-add>` which is available for everyone and does not check any authentication. When a HTTP post is invoked on a `<add-sputnik-page>`, permission is checked. If permission is granted, the constraints of the object type are checked. This calls the generic function `check(o :: <object>)` which guarantees uniqueness of usernames, etc.

`<detailed-view-sputnik-page>` first parses the object id after the last `'/'`, which is either a user name, a room name, or an IP address. This id is looked up in the data structure returned by `get-objects-of-page`. The thread variable `*object*` binds the object. Afterwards the dsp is processed.

`<edit-sputnik-page>` first checks permissions. Then it binds `*object*` to the requested object and processes the DSP. The post handler for edit pages checks permissions again and tries to change all slots of the object which have changed. It rolls back the change when the constraints are violated.

During processing of DSP files, HTML tags are encountered which may rebind the `*object*` thread variable, e.g. to each element of the list of observations of a specific tag. The simple design with one variable bound to the inspected object in the current context and another variable bound to the user was powerful enough for the HTTP interface.

The meta-programming facility was useful to counterfeit code duplication, especially to define multiple classes in this case. Also, multiple inheritance was used to inherit both from the datatype and from the operation of a page.

4.4 Main

After minor changes, the `<history-bot>` creates instances of `<sputnik-tag>` and stores them in the persistent object store each time it received an observation. At startup a XML file with observers was parsed and stored in the persistent object store.

A standalone application called `sputnik` started the services.

```
begin
  let reflector = make(<reflector>);
  let push-cl = curry(push-observation, reflector);
  make(<thread>,
      function: curry(reflector-top-level,
                      reflector));

  let udp-coll = make(<udp-collector>,
                    push-closure: push-cl);
  make(<thread>, function: curry(toplevel, udp-coll));

  let history-bot = make(<history-bot>,
                       push-closure: push-cl);
  add-client(reflector, hclient);
  make(<recurrent-timer>,
      interval: 5,
      event: curry(process-data, history-bot));

  make(<thread>, function: sputnik-web-main);
end;
```

It first instantiates a `<reflector>` and calls the listener loop in a separate thread. Then an `<udp-collector>` is instantiated with a closure pointing to the input of the reflector. The worker waiting for incoming packets is started in another thread. A `<history-bot>` is also instantiated and connected to the reflector, reporting every 5 seconds new observations. Finally, the web interface is started.

5. CONCLUSION AND FURTHER DEVELOPMENT

The whole project was an achievement, it was developed in about two months by teams which had not worked together and was successfully set up at the conference.

Dylan was a good choice for the middleware since it provides several abstractions which counterfeit source duplication but preserves high performance. It was also easy to integrate several changes in the communication layer. Also, the whole middleware was developed in the short time frame, eight weeks, by a single person as part time project.

In several places the code between separate libraries was reused. The whole middleware makes a lot of use of higher-order functions like `map`, `curry` and `reduce`. Also, the quite powerful meta-programming system of Dylan was used in several places, like binary parsing and the creation of classes for the HTTP interface. Multiple inheritance was useful for the page classes of the HTTP interface. Multiple dispatch was also used there and at other places.

Future development will include an automated way to interface XML schema definitions in Dylan, by writing a macro which transforms XML schema definitions to Dylan classes and transformers between those representations which check conformance of the schema.

The middleware had initially some bugs. A major performance problem was that all Ethernet frames were passed to userland and filtered there. Further `<udp-server-socket>` of the Unix sockets interface were broken. Thus explicit calls to C had to be used. The initial design lacked the queue for

clients which did not scale up because the thread creating an observation had to wait until it was delivered to all TCP clients. The author thanks Eric Blossom and John Gilmore for assistance in debugging.

The middleware has about 3000 lines of code, 1000 lines are the web interface. These are mostly definitions for HTML tags which are used in the dynamic server pages. More sophisticated data mining is planned for the next conference. This includes correlation with external social networks and public databases.

6. REFERENCES

- A. Bogk, H. Mehnert, Design and implementation of an object-oriented secure TCP/IP Stack, 23rd Chaos Communication Congress, 2006
- A. Shalit, The Dylan Reference Manual, Addison-Wesley, 1998,
<http://www.opendylan.org/books/drm/>
The Dylan programming language
<http://www.opendylan.org>
Sputnik source code
<svn://anonsvn.h3q.com/svn/sputnik/>
Koala <http://www.opendylan.org/koala/>
HTTP Authentication: Basic and Digest Access Authentication
- H. Mehnert, A. Bogk, A Domain-Specific Language for manipulation of binary data in Dylan, International Lisp Conference 2007, 2007