Helsinki University of Technology
Networking Laboratory
S-38.3138 Networking Laboratory, Special Assignment

DTN-based Blogging

Made by: Lauri Peltola
60965R
lauri.peltola@tkk.fi

Instructor: Jörg Ott
Supervisor: Jörg Ott

Submitted: 1.7.2007

Abstract

Network conditions in a mobile environment are often difficult from the point of view of traditional network protocols, such as TCP. Intermittent connectivity, long delays and nonexistent end-to-end paths hinder protocol performance or may prevent it altogether. A delay-tolerant network (DTN) is an overlay network that tackles these problems and allows communication to flow even in severe conditions. In this special assignment, we will devise an architecture for DTN-based blogging and also implement it in a concrete application. Traditional blogs require that a TCP connection from the user to the blog server exists. Our application allows a mobile user to send posts to her blog without a pre-existing end-to-end communication path. The underlying DTN takes care of delivering the post to the receiver, on an opportunistic hop-by-hop basis if needed. The user interfaces to the application—both for composing the posts and for reading them—are regular HTML pages, accessible with any web browser.

1. Introduction 4

2. Technologies and tools 5

2.1 Delay-tolerant networks 5

2.2 DTN2 6

2.3 Ruby 6

2.4 Dtn-ruby 6

2.5 Camping 6

2.6 SQLite 7

3. Overview of the application 8

3.1 Traditional blogs 8

3.2 Blogging through DTN 8

4. Implementation 9

4.1 Basic operation 10

4.2 Message format 11

4.3 Security 12

4.4 User interfaces 12

5. Conclusion 14

6. References 15

Appendix: Installation instructions 16

1. Introduction

The goal of this special assignment is to build a blog application on top of a delay-tolerant network (DTN). A DTN is an overlay network designed to operate in conditions in which common network protocols may fail. Typically this involves mobile nodes in a network where connectivity is intermittent.

The following chapters first introduce the concept of delay-tolerant networks and the various tools that are used in building the application. An overview of blogging in general is provided, along with a discussion on how adding DTN to the mix changes things.

Then, the actual implementation of the blog application is presented. First, we introduce the application architecture from a higher-level perspective, and then go through the operation and other aspects in more detail.

Finally, in a concluding chapter we evaluate the results of the work and discuss possible future improvements to the application.

2. Technologies and tools

The following chapters briefly introduce the technologies upon which the application is built, and the tools that are used in the implementation. The introduction to delay-tolerant networks below is based on [1].

2.1 Delay-tolerant networks

A delay-tolerant network is an overlay on top of a number of diverse regional networks, including the internet. Within a DTN, the regional networks may be extremely remote in terms of delay, and may employ, for example, different wireless technologies. The DTN overlay accommodates these varying network characteristics and provides a service that works regardless of “difficult” conditions in the underlying networks.

The motivation for DTN is that in certain situations the protocols used in the internet simply do not work. Examples of such situations are partitioned networks, highly asymmetric data rates, high error rates and long delays. A typical use-case for a DTN is an interplanetary network, e.g. a satellite orbiting Earth communicating with another satellite orbiting Mars. Within our solar system, propagation delays are in the order of minutes, and thus using a conversational protocol like TCP would be highly impractical.

DTN works by introducing a new protocol layer, the bundle layer, on top of the transport layer. The transport protocols used in the underlying regional networks need not be the same – the bundle layer is the glue that binds all the various lower layers together. The applications in the DTN only need to communicate with the homogenous bundle layer.

Bundles are messages that consist of the bundle header, control information (provided by the source application for the destination application) and user data. In essence, a bundle just extends the data encapsulation hierarchy with one further level.

The bundle layer has a set of mechanisms to overcome the difficulties of intermittent, long delay networks. The basic idea is to use store-and-forward message switching, i.e. hold bundles in a persistent storage along the communication path until the next hop comes available. An end-to-end path need not exist when the bundle is initially sent. Also, the bundle layer protocol is non-conversational in the sense that the nodes communicate between each other using simple sessions with minimal or no round-trips. Acknowledgments from the receiving node are optional.

To cope with long delays while still allowing TCP (or some other conversational protocol) to be used as the underlying protocol in some parts of the network, the bundle layer utilizes transport-layer termination. This means that a DTN node acts as a surrogate for a TCP end-node, isolating the TCP connection from the bundle layer.

DTN implements node-to-node reliability by means of custody transfers. In such a transfer, successive nodes in the bundle layer ensure the delivery of a bundle by means of retransmissions and acknowledgments. The bundle layer provides six different classes of service for bundle transmission. The source node may choose suitable service class depending on the nature of the transfer.

2.2 DTN2

DTN2 is a C++ implementation of the DTN bundle protocol by the Delay Tolerant Networking Research Group (DTNRG). [2] It is used as the basis of this work to provide the underlying DTN infrastructure.

2.3 Ruby

Ruby is the programming language in which our application is built. The Ruby website describes the language thusly: “Ruby is a dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write.” [4]

Ruby is an interpreted, object-oriented language that has a clear and concise, highly expressive syntax. Consider the following short example that defines a function which prints out the string “Hooray!” three times:

```
def say_hooray
  3.times { puts “Hooray!“ }
end
```

This piece of code demonstrates two of the many nice features in Ruby. First, in Ruby, *everything* is an object, including numbers and other primitive types. Because the number 3 is an object (an instance of the `Integer` class), it has its own set of methods, such as `times` in our example. Secondly, Ruby supports blocks (or closures) that can be attached to any method, describing how that method should act. Ruby also has a large standard library which has a wealth of functions for common tasks, e.g. socket programming.

2.4 Dtn-ruby

Dtn-ruby is a set of bindings to the application library of DTN2. [3] It provides an easy-to-use Ruby interface to the DTN2 functions, allowing the programmer to create applications that leverage DTN with minimal effort.

2.5 Camping

Camping is a tiny (less than 4 kilobytes in code) Ruby web framework that is used for creating the web interfaces of the application. It makes it very easy to create small applications that follow the model-view-controller (MVC) paradigm by providing the necessary, minimal infrastructure. Camping itself is very simple, but it also utilizes some external components, such as an implementation of the ActiveRecord pattern for object-relational mapping (ORM), and Markaby for markup creation with Ruby code. [5]

To be more specific, the following are the main features of Camping that are used in this work.

ActiveRecord: Blog posts—which are objects in the application code—are persisted in a SQLite database (i.e., blog post is a model, the M in MVC). ActiveRecord abstracts the relation between Ruby objects and database rows, allowing the programmer to manipulate the database without writing any SQL. For example, to retrieve a blog post with the title *Sake Bomb!*, one would do:

```
post = Post.find_by_title(“Sake Bomb!”)
```

And to save a new blog post:

```
Post.create({
  :title => “Cappin’ that Stat”,
  :body => “...text here...”
}
```

Routing: Camping allows easy routing of URLs to the application’s internal actions. Different actions can be run depending on the HTTP method (GET or POST). Consider the following:

```
class Index < R “/”, “/home”
  def get
     @posts = Post.find(:all)
     render :index
  end
  def post
     Post.create({:title => @input.title, :body => @input.body})
     render :post_created
  end
end
```

This piece of code defines a controller (the C in MVC), which responds to URLs `/` and `/home`. If the URL is requested with GET, all posts are fetched and displayed to the user. If the request is POST, a new post is created and the user is shown a corresponding message.

Markaby: The HTML views (the V in MVC) can be written in pure Ruby code using Markaby (Markup with Ruby). This approach is less verbose than plain HTML and makes it easy to mix variables within the views. For example, this is the layout for the blog frontend (`self << yield` adds view-specific content to the layout):

```
xhtml_strict do
 head do
   title 'dtn-blog frontend'
   link :rel => 'stylesheet', :type => 'text/css', :href => '/styles.css'
 end
 body do
   div.container! do
     h1.header { a 'dtn-blog frontend', :href => R(Index) }
     div.content { self << yield }
   end
 end
end
```

Alongside Camping we are using Mongrel, which is a slim and fast HTTP server for Ruby applications. [6]

2.6 SQLite

SQLite is a very lightweight, zero-configuration SQL database engine. It stores the entire database in a single file on the disk and supports most features of the SQL92 query language. [7] Because of its simplicity, SQLite is a perfect match for the Camping framework.

3. Overview of the application

The application that built for this special assignment is a blog. A blog is a website which displays posts written by the author in chronological order, often also allowing the viewers to add their own comments. Blogs usually focus on one subject which can be virtually anything that people can be interested in: technology, music, photography, wine, and so on. Some blogs are the author's personal diaries, used for keeping up with friends and family, or just for the pleasure of writing and sharing.

The power of blogs has also been noticed in the mainstream – large newspapers have blogs on their websites as a complementary service, and many companies use blogs for corporate PR.

3.1 Traditional blogs

The usual blog setup is such that the application is hosted on a web server along with a database that holds the content. The author uses a web browser to connect to the server over HTTP and typically authenticates with a username/password combination. After that, an HTML form interface is used to compose the content, which is then sent over HTTP to the server and saved to the database. The blog can be read by anyone with a web browser. [1]

Because of the reliance on HTTP, all this assumes that TCP connectivity is available. If this is not the case, the blog cannot be used. A person trying to create an entry to her blog while on the move with a mobile device would have to save a local copy of the post and wait until the network conditions allow an HTTP connection to the web server.

3.2 Blogging through DTN

Our blog application differs from the traditional blog in the way in which new entries are posted on the blog. The back-end side of things is the same: a web server hosts the blog application and the content database. The difference is that in our architecture, the author does not use HTTP to connect to the server whenever she wishes to create an entry – the interface for composing posts exists on the (mobile) device she is using. The post is composed locally and formatted into a bundle, which is then sent through DTN to the back-end server.

This means that TCP connectivity is not required. Only an arbitrary DTN infrastructure, somehow connecting the mobile device with the web server, must exist.

[1] It is common that blogs also offer RSS/Atom feeds, which allow reading the content without actually using a web browser to navigate to the site. Nevertheless, the mechanism for retrieving the content is the same: an HTTP GET request.

4. Implementation

Figure 1 below depicts the architecture of the application. The main parts are the blog front-end, the blog back-end and the blog daemon.

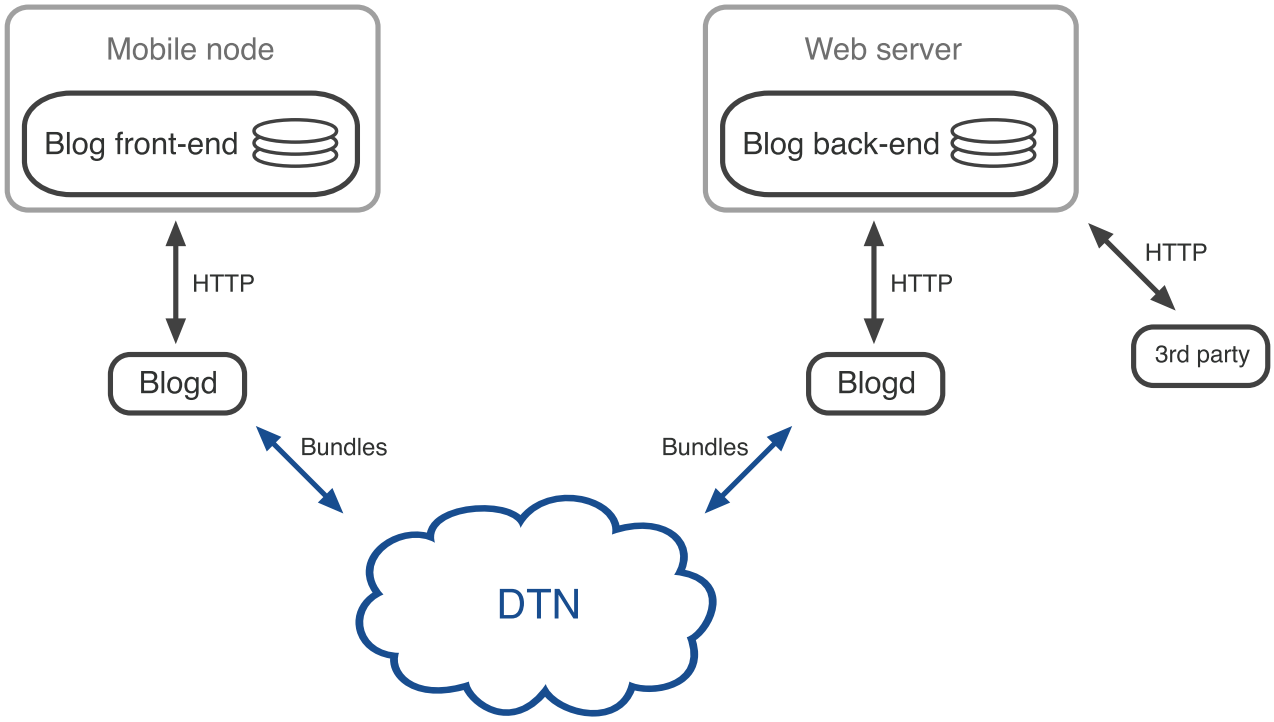

Figure 1: The architecture of the application concept.

The blog front-end is the part of the application that runs on the mobile node and is used to compose the blog posts. The front-end also has persistent storage (a database) in which information about sent posts is saved.

The blog back-end is where the posts end up after being delivered through the DTN. It runs on a web server and can be accessed via HTTP by anyone who wishes to read the blog. The back-end also needs database in which to store the blog posts.

The blog daemon (Blogd in figure 1) is an entity that exists on both the front-end and back-end sides of the blog. On the back-end side it listens to incoming bundles from the DTN, forwards them as HTTP requests to the blog and sends acknowledgment bundles back to the front-end.

On the front-end side the daemon listens to incoming HTTP requests from the mobile node, formats them into bundles and sends them through the DTN. The daemon also receives acknowledgment bundles and forwards these as HTTP requests to the mobile node.

The blog daemons may be co-located with the front-end and back-end nodes, but that need not be the case. Since the communication between the daemons and the end-nodes is standard HTTP, it may go through an arbitrary path.

In our implementation, both the front-end and back-end are web applications built on the Camping framework. Thus, composing blog posts is done the traditional way, with a web browser using HTML forms. It is not a requirement that the front-end is a browser-accessible web application, it might as well be a native application on, for example, a Symbian mobile device.

The blog daemons are not separate physical entities on either end – they are running on the same system as the Camping applications. SQLite databases are used for persistent storage.

4.1 Basic operation

Figure 2 shows a more detailed view of the architecture as it is realized in our implementation. The addresses for the different interfaces are shown. As mentioned above, the blog daemons are not separate entities, which means that the HTTP traffic confined within the end-systems. Only the DTN traffic flows between the systems, through some transit network. [2]

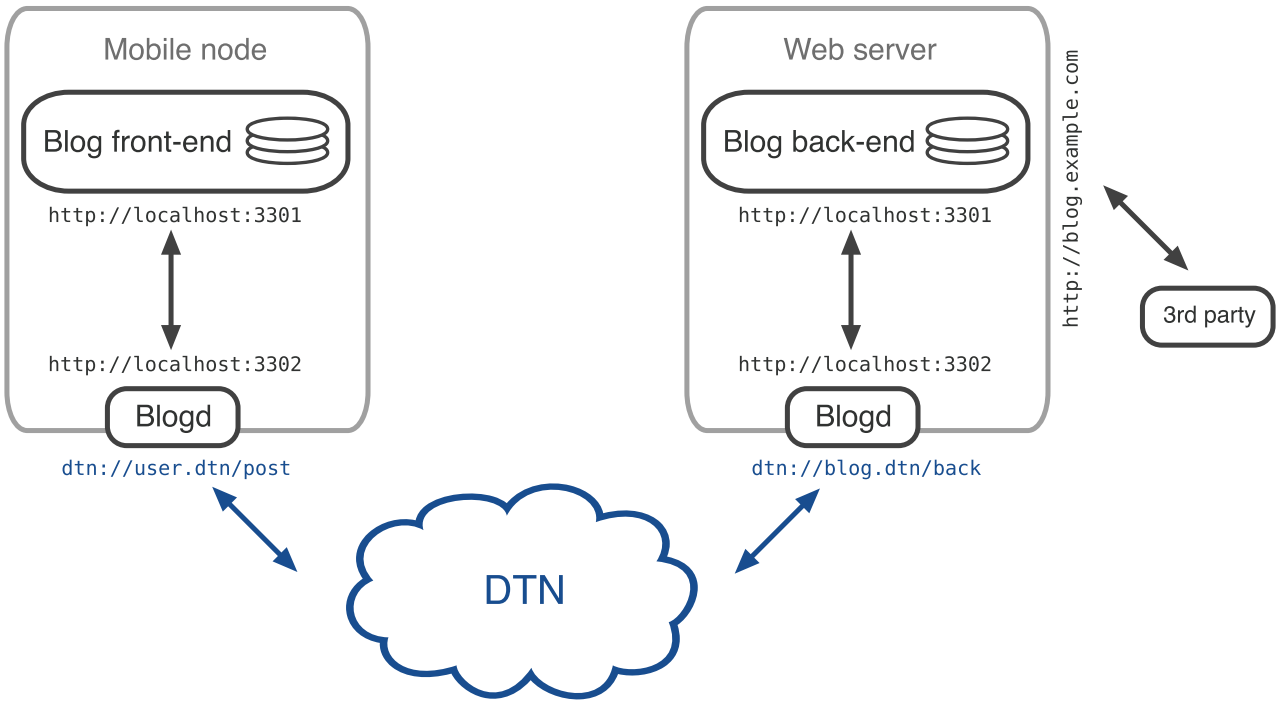

Figure 2: The architecture of our implementation.

The following example illustrates the operation of the application step-by-step:

1. The user of the mobile node (e.g., a laptop computer) fires up a web browser and opens the blog front-end page. She is presented with an familiar looking HTML form found in all web based blogs. She types in the title of the post, the actual text and attaches an image file.

[2] Actually, in our development setup, *all* the entities are within a single computer and no traffic flows through any network. However, if this system were deployed, it would work as described in the text.

2. The blog front-end application formats the post into a S/MIME multipart message and sends it as an HTTP POST request to the blog daemon at `http://localhost:3302`. Also, an entry for this post is saved to the local database.

3. The blog daemon receives the post over HTTP, takes the body of the request and puts it into a bundle, and sends the bundle to `dtn://blog.dtn/back`.

4. The bundle travels through the DTN to the other end, where the back-end blog daemon receives it. It puts the bundle payload into an HTTP POST request and sends it to the blog back-end at `http://localhost:3301`.

5. The blog back-end application receives the HTTP request, parses the S/MIME message, saves a new post to the database and returns 200 OK. If the verification of the digital signature in the post fails, 403 Forbidden is returned.

6. The blog daemon sends back an acknowledgment bundle containing the 200 OK response code.

7. The front-end daemon receives the acknowledgment bundle and forwards it as an HTTP request to the front-end application.

8. The blog front-end receives the response code and knows that the post was successfully delivered. It updates the local database entry for the post, after which the user can see on the web interface that the post has gone through.

4.2 Message format

The posts are formatted nested S/MIME multipart messages. The “outer” message contains two parts: user data and a digital signature (explained further in the following chapter). The user data itself is a MIME multipart message.

In the user data part, the Message-ID header is used for uniquely identifying the posts. The message has two mandatory parts, post title and post body, always in the this order. A fourth optional part may exist, containing an image file attachment (encoded in base64 format). The following is an example of the S/MIME post format:

```
MIME-Version: 1.0
Content-Type: multipart/signed; protocol="application/x-pkcs7-signature"; mi-
calg=sha1; boundary="----E87A703893D676294B5733282BE4AFC7"

This is an S/MIME signed message

------E87A703893D676294B5733282BE4AFC7
Message-ID: 30
Content-Type: multipart/mixed; boundary="=-1181845538-996536-14870-1398-1-="
MIME-Version: 1.0


--=-1181845538-996536-14870-1398-1-=
Content-type: text/plain; charset="utf-8"

This is the title of the post.
--=-1181845538-996536-14870-1398-1-=
Content-type: text/plain; charset="utf-8"
```

```
And this is the body.
--=-1181845538-996536-14870-1398-1-=
Content-type: image/jpg
Content-transfer-encoding: base64

[FILE DATA]
--=-1181845538-996536-14870-1398-1-=--

------E87A703893D676294B5733282BE4AFC7
Content-Type: application/x-pkcs7-signature; name="smime.p7s"
Content-Transfer-Encoding: base64
Content-Disposition: attachment; filename="smime.p7s"

[SIGNATURE]
------E87A703893D676294B5733282BE4AFC7--
```

4.3 Security

Blog posts are, by definition, meant to be seen by the public. Thus, there is no reason to protect the confidentiality of the posts by means of encryption. However, authentication and data integrity must be ensured, i.e. the back-end server must be able to verify the identity of the sender, and that the content of the post has not been tampered with along the way.

For this purpose, S/MIME digital signatures are used. When sending a post, the front-end signs the message with a certificate and 2048-bit private key. The back-end server verifies the signature before accepting the post. If the verification fails, the server responds with HTTP status 403 Forbidden. This allows the blog daemon to report the error back to the sender.

The certificate and private key are stored within the front-end in PEM-format.

4.4 User interfaces

There are two user interfaces for the blog: one for composing the posts and another for reading them. These are regular HTML pages, generated by the Camping applications and served by the Mongrel web server. Not much effort has been put into designing the looks of these interfaces, as that is not the point of this work.

Figure 3 shows the public interface of the blog, which anyone can access. It is no more than a list of posts, with each post having a title, a date of publication, a body text and an optional image attachment. The figure also shows a link that can be clicked to delete the post, which, of course, wouldn't be available for the public.

Figure 4 shows the interface that is used for composing posts. It is a normal HTML form with fields for the title, body and image attachment. There is also a list of previously sent posts and their statuses. The status of a post can be one of the following:

- Sent – Post was sent, no response has been received.
- Delivered – Post has been acknowledged.
- Unauthorized – Post was received but signature validation failed.
- Failed – Post was received but could not be parsed.

Figure 3: The public interface for reading posts.

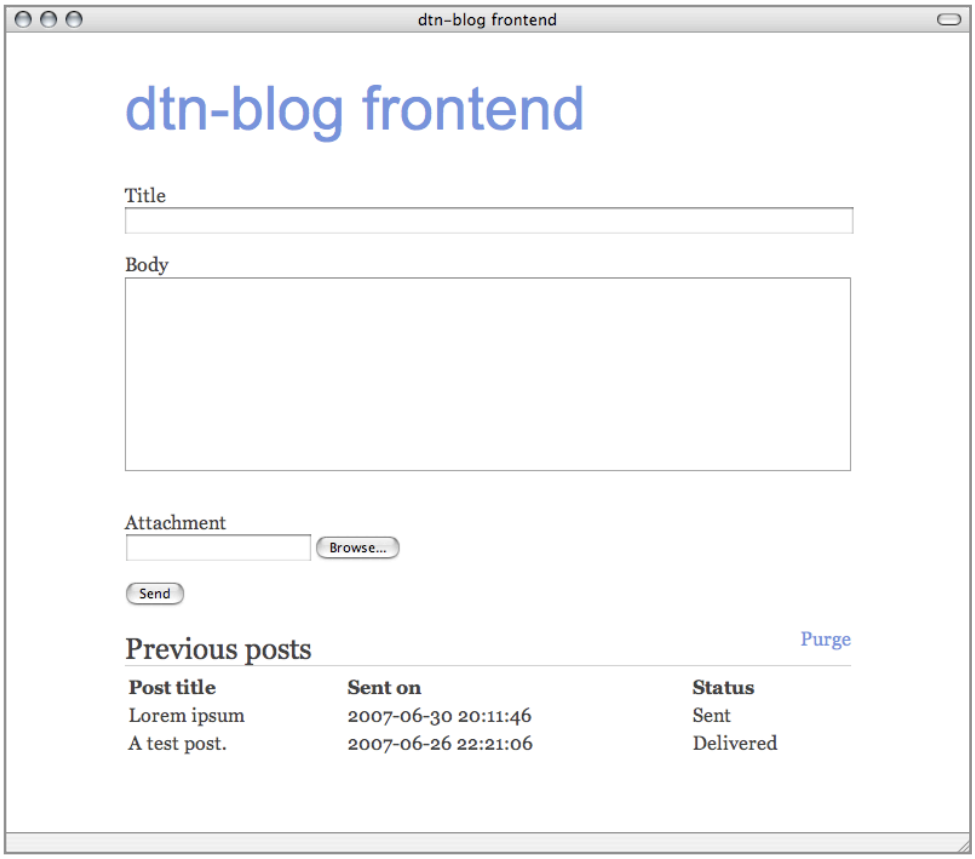

Figure 4: The interface for composing posts on the mobile node.

5. Conclusion

During the course of this special assignment, the concept of blogging through the DTN has proven to be an idea to pursue. DTN mitigates the issues that are related to network connectivity of mobile nodes, especially the problems that hinder the performance of TCP, thus making web access difficult. Using the DTN infrastructure, the application can overcome the typical problems associated with mobility: intermittent connectivity, long delays, high error rates and such.

The blog application we built is a functional demonstration of the concept. The application works as such, and could be used if a DTN infrastructure were available, although for real-world usage the installation procedure (described in the appendix) should be slimmed down to produce an easy-to-use software package.

One should also note, that it is not necessary to build a custom back-end application for the blog, as we have done. It would certainly be possible to use one of the popular, freely available blogging engines as the back-end. Only the blog daemon that translates bundles to HTTP, and vice versa, would have to be changed to accommodate the blogging engine's internal HTTP interface. Developing such a "plugin" would surely be a feasible task.

Presently, DTN is used only for sending the posts to the blog. To read the blog, a plain old well connected web browser is still needed. Developing functionality to allow the blog also be read through DTN would the obvious next step. A blog is, in a hierarchical sense, a rather simple structure. Thus, bundling a blog (or a part of it) and sending it through DTN should not be too difficult. However, this problem can be augmented to the general case of sending arbitrary web content over DTN, which is a whole subject of its own.

6. References

[1] Forrest Warthman: Delay-Tolerant Networks (DTNs): A Tutorial (2003)

[2] Delay Tolerant Networking Research Group (www.dtnrg.org)

[3] Dtn-ruby (prj.tzi.org/cgi-bin/trac.cgi/wiki/Dtn-ruby)

[4] Ruby language (http://www.ruby-lang.org/en/)

[5] Camping (http://code.whytheluckystiff.net/camping/)

[6] Mongrel (http://mongrel.rubyforge.org/)

[7] SQLite (http://www.sqlite.org/)

Appendix: Installation and usage instructions

Installation

The following steps will guide you through installing the software that is required to run the blog application on UNIX-based environments. Note that DTN2, Ruby and Dtn-ruby have additional prerequisites that—depending on your operating system—might or might not be installed already. You will need to install these dependencies first if they are not present. If necessary, consult Google for help.

1. **DTN2** – Follow the installation instructions found on the DTNRG website (`http://www.dtnrg.org/docs/code/DTN2/doc/manual/compiling.html`) to obtain and compile DTN2.

2. **Ruby** – Download the latest Ruby source code from `http://www.ruby-lang.org`. GNU-readline is needed for the Ruby console (`irb`), but you may also compile without readline support. Untar the file, configure and compile as per usual:

   ```
   ./configure --prefix=[/your/path/of/choice] --enable-pthread
               --with-readline-dir=[/path/to/readline] --enable-shared
   make
   make install
   ```

3. **Rubygems** – Rubygems is a package manager for Ruby. It will be used to install the various components (gems) used by the application. Obtain the latest source from `http://rubyforge.org/frs/?group_id=126`, untar, and install by running `setup.rb`:

   ```
   ruby setup.rb
   ```

4. **Gems** – Installing gems is a simple one-line task. The required gems are Mongrel, Camping and Rubymail. To install, do the following (and select the latest Ruby version of each gem):

   ```
   gem install mongrel --include-dependencies
   gem install camping --include-dependencies
   gem install rmail
   ```

5. **Dtn-ruby** – Installation instructions for dtn-ruby can be found on the project website (`https://prj.tzi.org/cgi-bin/trac.cgi/wiki/Dtn-ruby`). The source code is only available as a Subversion checkout, so you will need the `svn` command line tool.

6. **The blog application** – Now you have the necessary infrastructure installed. You will also need a certificate and a private key for signing the S/MIME messages. You can use the ones included in the tarball or create your own with a suitable application. The certificate and key must be in PEM format, and the key must use the RSA algorithm. Finally, untar the blog application to an appropriate location and you are set.

Usage

You may run the front-end and back-end applications on the same computer or on two separate computers. In the latter case, you have to set up DTN routing between the two systems. These instructions assume that you are using just one computer for both ends of the application. However, using two computers is no different, just run the necessary commands on both systems.

1. Start the DTN daemon.

   ```
   dtnd
   ```

2. Start the Camping applications.

   ```
   camping blog_* -d blog.db
   ```

   This will mount the applications to `http://localhost:3301/blog_backend` and `http://localhost:3301/blog_frontend`.

3. Start the back-end blog daemon, specifying the local DTN endpoint id for the back-end, and the HTTP address to which the posts are forwarded.

   ```
   ruby blogd.rb dtn://blog.dtn/backend http://localhost:3301/blog_backend/add
   ```

4. Start the front-end blog daemon, specifying the local DTN endpoint id for the front-end, the HTTP address to which the acknowledgments are forwarded, and the remote DTN endpoint id of the back-end.

   ```
   ruby blogd.rb dtn://blogger.dtn/post …
   … http://localhost:3301/blog_frontend/acknowledge dtn://blog.dtn/backend
   ```

5. Point your web browser to `http://localhost:3301/blog_frontend` and make a test post. Go to `http://localhost:3301/blog_frontend` and see if it worked.