

# **1 Non-web use for Apache/mod\_perl: SMS app**

## 1.1 Bas A.Schulte <bschulte (at) zeelandnet.nl> exclaimed:

- Date: Fri, 22 Mar 2002 14:01:28 +0100

### Preface

This is a story about how about I've used a combination of perl, Apache and mod\_perl to create a component-based service architecture that implements a platform for building SMS applications. By reusing capabilities offered by Apache/mod\_perl I saved a lot of time developing the system. The strong OO features of perl that I used enabled me to build a very flexible system as well to cope with future requirements. We had the platform in place in about 6 weeks, starting with absolutely nothing: no hardware, no development environment, no technology choices made beforehand.

### Introduction

The purpose of the system to be developed was to provide a server platform on top of which arbitrary SMS (Short Message Service) applications can be developed quickly. It should be built using a stable and scalable architecture with room for future enhancements such as integrated billing and reporting options.

An SMS application can be characterized by subscribers sending text-based commands to the platform and have the platform dispatch to the right application instance. The application instance handles the command, executing whatever application-logic defined by that particular application, and usually generate one or more responses. It should also be possible that the platform initiates messages to subscribers as a result of a request sent by another subscriber as well as be able to generate messages based on timers

There also was a requirement to have the framework publish application-specific data in XML to allow customers to display this data on other media channels such as a website.

Connecting the platform to external entities for the transmission and reception of SMS messages such as SMSC's (SMS Centers distribute SMS messages to and from mobile subscribers) and SMS Gateways (smart front-end to one or more SMSC's unifying the method to reach subscribers from multiple telecom operators) should be flexible enough to be able to "plug-in" different protocols such as HTTP/SMTP/CIMD/SMPP as needed.

### Component architecture

Early on in the project I decided to go for a distributed component architecture. Individual components should be deployable on multiple physical machines. This offers the required scalability and the ability to define a convenient security scheme by running components on segments of a network with differing outside visibility requirements.

As I started modelling this "world", I ended up with the following

components:

#### 1. Application server

Within this application server, multiple instances of multiple SMS application instances should be running. The actual application-logic is running within this component. This component provides two external services:

- handleMessage(CommandRequest)

This service takes an instance of a CommandRequest object and runs the command in the appropriate application instance.

- handleTimer(Timer)

This services handles expiry of a timer set by the application-logic of an SMS application.

- getView

This service allows a client to retrieve application-defined views in XML.

#### 2. Timer service

A persistent service that maintains timers set by application instances within the game application server and invokes the

handleTimer service of the game application services upon expiry of a timer.

External service offered:

- setTimer(Timer)

#### 3. Virtual SMS gateway (VSMSC)

This component handles communication with the outside world (the external entities such as SMSC's and SMS gateways). This component is split up in 2 subcomponents, one that handles input from mobile subscribers and one that handles output to mobile subscribers. Each subcomponent provides one service:

- handleMessage(Message)

The input component receives requests from the outside world using pluggable subcomponents that handle protocol details, the output component transmits requests to the outside world using pluggable subcomponents that handle protocol details.

#### 4. XML Views service

This component offers an HTTP interface to retrieve

1.1 Bas A.Schulte <bschulte (at) zeelandnet.nl> exclaimed:

application-specific views in XML. It uses customer-specific XSLT stylesheets to transform the XML data. This component is largely based on Matt Sergeant's AxKit. AxKit allow the source of your "document" to be delivered by your own provider class by subclassing off of AxKit::Provider. My provider class talks to the application server's getView service while AxKit performs its miracles with all kinds of transformation options.

Components Figure 1 System components

Apache/mod\_perl as a component container

When thinking about how to implement all this I was tempted to look into doing it with some J2EE-thingy. However, there was this time-constraint as well as a constraint on available programmer-hands: I had one freelance programmer for 20 days and I had to arrange the whole physical part (get the hardware, a co-location site etc.). Then it struck me that this application server really looked like a vanilla regular mod\_perl web application: receive request from user, process, send back reply. No html though, but Message objects that could be serialized/deserialized from text strings. There were of course some differences: the reply is not sent back inline (i.e. upon reception of a request via SMS, you can't "reply"; you have to create a new message and send that to the originator of the request) and there also was the timer service: I can't make Apache/mod\_perl do work without having it received a user-initiated request.

The good thing was I've been doing Apache/mod\_perl for some years now so I knew beforehand I could create a schedule acceptable from the business point of view that was also feasible based on experience with the technology.

So, for each component except the timer service, I defined separate Apache/mod\_perl instances, one for the application server, one for the SMS output component, one for the SMS input component and one for the XML Views component.

Each instance defines a URL for each service that the component running in the instance provides.

Component communication

I took a shortcut here. I wanted to go for SOAP here as it seems a natural fit. It will allow me to move components to other languages (management and marketing still seems hung up on java) fairly easy. My personal experiences with SOAP on earlier projects weren't too good and I just couldn't fit playing with SOAP into my schedule. So I took my old friends LWP::UserAgent, HTTP::Request and Storable to handle this part (perl object instance -> Storable freeze -> HTTP post -> Storable thaw -> perl object instance).

The good thing is that this actually is a minor part of the whole system and I know I can put SOAP in easily when the need arises.

"Breaking the chain"

I did make one mistake in the beginning: all service calls were synchronous. The initial HTTP request would not return until after the whole chain of execution was done. With possibly long running actions in the server component, this was not good. I had to find a way to execute the actual code *after* closing the connection to the client. Luckily, Apache/mod\_perl came to the rescue. It allows you to set a callback that executes after the HTTP responses are sent back to the client and after it closes the TCP/IP connection.

Result

We had the platform in place in about 6 weeks, starting with absolutely nothing: no hardware, no development environment, no technology choices made beforehand. Based on former experience, the decision to go with a LAMP architecture (Linux, Apache, MySQL, Perl) running on fairly cheap intel boxen was made quickly. MySQL was, and is, not on my wishlist, but the whole battle of moving Oracle in would have been both a time as well as a money killer, either of which we didn't have a lot of at the time.

Aside from having one production SMS application (a mobile SMS game), I've done a prototype SMS application on this platform to check if it really is easy to create new apps. It took me about 4 hours to implement a "SMS unix commandline" application: I can login to the application server using SMS, send Unix commands with my mobile phone and receive their output (make sure your command doesn't generate more than 160 characters though). The application also maintains state such as the working directory I'm in at any given time.

Performance is 'good enough' with the platform running on 2 fairly cheap Intel boxen, it handles 40 to 60 incoming request per second. As I haven't spent one second on optimization yet (anyone know the command to create an index in MySQL?), that number is fine for me. I did put 1 gigabyte in each machine though as the Apache child

processes eat up quite some memory.

Future enhancements and considerations  
SOAP

I really want SOAP. It just seems to make sense to do so: it was invented for doing stuff like this and I like the concept of WSDL. It allows you to define the interface in an XML file so clients "know" what type of parameters the service needs as well as the return parameter types.

SOAP will also allow new components that are not perl. SOAP is available in a lot of languages and integration of the various SOAP implementations is getting better every day (see here ).

1.1 Bas A.Schulte <bschulte (at) zeelandnet.nl> exclaimed:

#### Framework for service-based architecture

I'd like to extract the code that handles the communication between the components in the current system and create a generic framework that allows one to easily create an Apache/mod\_perl-based components container. The available services would be registered in httpd.conf and there should be a service-discovery mechanism. On the client side, I'm thinking about something that makes it easy to create client-side stubs. Stay tuned...

#### Apache/mod\_perl 2.0

This looks very promising to create generic components containers. It is very easy to create non-HTTP based services with Apache 2.0 with mod\_perl's 2.0 support for writing protocol modules in perl. Also, the various multi-process models (most notably threading) available in Apache 2.0 should result in better performance or at least more choices as far as the process model is concerned.

#### Lamp

I'm still a little unsure about LAMP. Can we move to relatively cheap hardware and a free OS when we were used to (very) expensive HP, Sun or IBM hardware and get away with it? Personal experience and what I've read from others seems to indicate we can. Experience will tell, and if it breaks, moving the platform to either of the above three should be a no-brainer. We live in interesting times.

## Table of Contents:

1	Non-web use for Apache/mod_perl: SMS app . . . . .	1
1.1	Bas A.Schulte <bschulte (at) zeelandnet.nl> exclaimed: . . . . .	2