A Programmable Client-Server Model: Robust Extensibility via DSLs

Charles Consel Laurent Réveillère

INRIA/LaBRI
ENSEIRB 1, avenue du docteur Albert Schweitzer,
Domaine universitaire - BP 99
F-33402 Talence Cedex, France
E-mail: {consel,reveillere}@labri.fr

Abstract

The client-server model has been successfully used to support a wide variety of families of services in the context of distributed systems. However, its server-centric nature makes it insensitive to fast changing client characteristics like terminal capabilities, network features, user preferences and evolving needs.

To overcome this key limitation, we present an approach to enabling a server to adapt to different clients by making it programmable. A service-description language is used to program server adaptations. This language is designed as a domain-specific language to offer expressiveness and conciseness without compromising safety and security. We show that our approach makes servers adaptable without requiring the deployment of new protocols or server implementations.

We illustrate our approach with the Internet Message Access Protocol (IMAP). An IMAP server is made programmable and a language, named Pems, is introduced to program robust variations of e-mail services.

Our approach is uniformly used to develop a platform for multimedia communication services. This platform is composed of programmable servers for telephony services, e-mail processing, remote-document processing and stream adapters.

1. Introduction

The *client-server* model is a software architecture commonly used to support a family of services in the context of a distributed system. A server implements a set of services. Clients, connected to the server's system with a network, send requests to access these services. When the server has processed a request, it sends a response to the corresponding client.

The rules and conventions used by the server and a client to interact are defined by a *protocol*. A protocol has two main purposes:

A specification for implementers. It precisely defines both the formats of data and the requests/responses that can be exchanged between the server and a client. As such, a protocol provides guidance for an implementer of the server or a client to develop all the required functionalities. This specification is detailed enough so that a client can be implemented independently of a given server. In fact, although the server has to provide the complete family of services, a client can only implement a subset of these services.

A definition of a family of services. A protocol defines a family of services implicitly, in that, determining this family requires a careful study of the protocol requests and responses. Such study allows one to identify the required family of services to be provided by the server, and subsets of the family of services to be implemented by a client.

To illustrate the client-server model, consider a family of services for a remote access to mailboxes. This family of services aims to provide a client with access to messages stored on a server, at some possibly distant location. In fact, there are many ways in which these services can be realized. One instance is defined by the Internet Message Access Protocol (IMAP) [11, 20]. There exist various implementations of IMAP servers. There is an even greater variety of IMAP clients, ranging from simple e-mail reading tools targeted toward embedded systems (*e.g.*, Althea [2]) to integrated Internet environments (*e.g.*, Microsoft Outlook [19], Netscape Messenger [21]) for workstations. As such, IMAP illustrates how much, for a given protocol, clients and servers can be independently implemented.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

1.1. Why is the Client-Server Model Limited?

Although successful, the client-server model is limited in that it offers little, if any, sensitivity to the client needs and requirements. Indeed, in this model, the server consists of a fixed implementation of a set of services which greatly limits its ability to adapt to a client. Let us review instances of this insensitivity and illustrate them with the IMAP case.

- *Insensitivity to client terminal capabilities* (*e.g.,* audio, video, computing power, energy consumption, ...). Insensitivity to the display capabilities of the client terminal may cause the server to deliver inappropriately formatted data to the client. For example, a colored message body, sent to a black-and-while display, is a waste of bandwidth and computing power.

- *Insensitivity to network features* (*e.g.,* available bandwidth, billing policy, ...). Insensitivity to network bandwidth makes it impossible for the server to adjust the volume of information sent to the client when needed. For example, when the available bandwidth is low, a high-quality audio file (*e.g.,* 16-bit stereo sampled at 44,1 kHz), attached to a message, should be degraded to avoid overloading the network.

- *Insensitivity to client preferences.* This insensitivity causes all clients, regardless of their client terminal, to have the same view on their mailbox. For example, when a client wants to access the messages of a mailbox, the server sends an exhaustive summary of all messages. There is no mechanism to minimize this summary with respect to some user-defined filters when this request comes from a client terminal with limited capabilities (*e.g.,* a cell phone).

- *Insensitivity to rapidly evolving market needs.* Fierce competition among telecommunication companies and hardware manufacturers, compounded with highly volatile user trends, should result in rapid service development and deployment. However, the services offer by a server are frozen by a protocol and thus cannot keep pace with market opportunities.

As illustrated by the IMAP case, the insensitivity of the client-server model can be a major limiting factor for its applicability to fast changing requirements and needs. This insensitivity has long been identified and solutions has been proposed.

The first common approach to remedying this deficiency consists of customizing an existing protocol with respect to the requirements of a new usage context. This approach leads to a proliferation of protocols which causes obvious compatibility problems. Consequently, it works against major standardization efforts most notably made by the telecommunication and networking communities.

Recent protocols, like RTSP [25], and new versions of traditional ones, like HTTP/1.1 [10] acknowledge the fact that a protocol should cope with evolving needs and that it should be sensitive to client requirements. Their strategy to address this issue mainly consists of offering three possible extensions: new headers in requests/responses, new error codes and, new requests. Again, this strategy defeats the idea of standardizing interactions between a server and, independently developed, clients. Besides, it is a server-centric approach whereas adaptability often needs to be defined with respect to the client.

Another approach consists of enabling code to be introduced on the server side in the form of scripts (*e.g.,* CGI scripts [4]) parameterized with respect to some client data. Again, this strategy is server-centric because scripts can only be introduced by the server administrator. Consequently, it is limited to the scope of adaptability foreseen by the owner of the server.

An alternative aims to leave the server unchanged but to rely on the client to adapt its behavior to the needs and requirement of the user. In this strategy the server remains insensitive to the client, with the drawbacks mentioned earlier; the client devotes computing power and time to adjust to the user's needs. This strategy is illustrated by the latest version of Mozilla (version 1.3 Alpha) [16] where messages are being processed locally to fit the client preferences.

A related approach consists of introducing a proxy server that runs client scripts and invokes the unchanged server. Because the client scripts are *a priori* untrusted by the server administrator, the proxy needs to run as a different process, or even, on a different machine. In this context, not only does a request trigger computations in the server, but it also requires the proxy to execute the client script to process the response. In addition, if the proxy and the server run on different machine, they consume bandwidth to communicate.

None of the above approaches satisfactorily allow client needs and requirements to be propagated to the server. They are limited to addressing the needs and requirements expected by the protocol developer.

1.2. Our Approach

To make the client-server model sensitive to client requirements and needs, we propose to make the set of services supported by a server programmable. To do so, our strategy consists of enabling a client (or a third-party developer) to provide the server with a specific implementation for the processing of a request. By varying the implementation of a request, one can vary the definition of a specific service. As an example, in the IMAP case, a client could re-

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

define the request aimed to list the messages of a mailbox to only report on the ones coming from a particular domain, so as to minimize the volume of information.

Yet, providing arbitrary implementations for a request obviously compromises the robustness of the server: one could attempt to upload erroneous, inappropriate, or even, malicious implementations. To alleviate these risks, we propose to introduce a *service-description language* to specify new services. Following the generative programming approach [8], service implementations are *automatically* generated from their description.

A service-description language is a domain-specific language (DSL) in that it offers appropriate abstractions and notations, and it is restricted such that critical properties can be checked [6]. Indeed, making specific properties decidable is a key parameter in the design of most DSLs. Examples of DSL properties include linear usage of resources [18], termination [6], and deadlock-free schedulers [3]. These properties often go beyond the scope of general-purpose checking techniques, whether dynamic, such as sand-boxing [29], or static, such as type checking [1]. Because such properties are undecidable in general, in the context of general-purpose languages (GPLs), both static and dynamic program analysis produce unpredictable results.

DSLs have been studied in the context of various application domains for many years and have shown benefits in terms of expressiveness, conciseness, safety and performance [28]. We argue that, just like DSLs are a well-recognized solution to address families of programs [6, 28], they can be uniformly used to model families of services with similar benefits.

We propose a systematic approach to making servers programmable. This approach is based on a software architecture for servers where each request is a potential point at which programmability can be introduced. The scope of programmability corresponds to the scope of the services which can be designated. Some properties, critical to a family of services, can be guaranteed by definition of the DSL (*e.g.,* termination and predictable resource usage).

1.3. Contributions

Most families of services are bound to evolve, often rapidly and unpredictably in emerging domains such as multimedia communications. To address this key issue, our contributions can be summarized as follows.

- We identify where the client-server model can be made adaptable, namely, in the treatment of requests. This choice is motivated and illustrated.
- We demonstrate that making requests programmable allows a service to adapt to unforeseen needs and requirements, without deploying new protocols and server implementations.
- We show that DSLs enable server programmability to be controlled and disciplined without compromising robustness.
- We illustrate our approach with the IMAP protocol and an IMAP server. The server has been modified to make it programmable; new services are safely defined in a DSL called Pems.
- Finally, we argue that combining programmable servers leads to rich, yet robust services. This combination is illustrated by the IMAP programmable server combined with a new server aimed to process documents remotely.

Our approach is uniformly used to develop a platform for multimedia communication services. This platform, named Nova, consists of programmable servers to define telephony services, e-mail services, remote-document processing services, and stream adapters.

Overview

Section 2 introduces the notion of a programmable client-server model. Section 3 describes how to develop a programmable server from a given protocol. Section 4 shows how it scales up to a complete platform for communication services. Section 5 assesses our approach based on some experimental data collected in the context of the IMAP case. Section 6 gives concluding remarks and discusses future work.

2. The Programmable Client-Server Model: What

In this section, we present the main stages of our approach to introducing the notion of programmability in the client-server model. In this approach, a developer first determines variations for the family of services underlying a protocol definition. Then, he studies the mapping of the identified variations into the protocol by determining where programmability requires the protocol to be generalized.

2.1. Service Variations

This study of service variations is done from the client's viewpoint, to address his needs and requirements. A starting point consists of collecting existing variants of a protocol and integrating them as variations of generic services. A complementary strategy aims to extrapolate on technological evolution.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

Yet, in contrast with protocols, our approach is not aimed to exhaustively determine potential variations. Rather, it delimits a scope of variations. Specific points within this scope will later be designated by particular DSL programs.

In the IMAP case, our goal is to identify a set of variations characterizing a scope of customized accesses to a mailbox. We explore these variations systematically by considering the various levels involved in accessing a mailbox, namely, an access-point, a mailbox, a message, and its fields (*i.e.,* message headers and parts). At each level of this hierarchical schema, we study what programmability could be introduced. We refer to the programmability of each level as a *view*.

- At the top-level, an access-point view should define coarse-grained parameters such as the client terminal features, the characteristics of the link layer, and a mailbox view.
- A mailbox view should enable one to only consider messages which go through a user-defined filter. This should prevent information from flooding a limited client terminal. Each retained message should be assigned a message view for further customization.
- A message view should define the layout of a message, that is, the fields involved in a message. It should drop fields that are irrelevant with respect to a given message view. Each retained field should be assigned a field view for specific processing.
- A field view should define the layout of a field value, that is, the value of a message header, a message attribute or a message part (*e.g.,* the `From` header, the total message size, and the message body, respectively). It should appropriately format field values with respect to the client needs and requirements. This process should include such treatments as erasing voluminous values, condensing values, and converting the format of field values.

2.2. Mapping Service Variations into a Protocol

Once the variations have been identified, the protocol can be examined to determine how to map these variations into the protocol. A protocol defines the requests/responses exchanged between the client and the server. Each request is an abstraction corresponding to a given service (or a part of it).

The service variations, identified in the previous phase, need to be associated with specific requests. Different kinds of association can occur between variations and requests. Let us illustrate this phase with the IMAP example.

Some requests may be out of scope with regard to the service variations previously identified. For example, IMAP manages multiple mailboxes and thus offers requests (*a.k.a.* commands) to create and delete a mailbox. None of the identified service variations are concerned with these requests.

A service variation may impact several requests. The number of recent messages, for instance, is part of the response of three requests (*i.e.,* `Select`, `Status` and `Examine`). These requests are thus affected by a mailbox view since they should now only report on recent messages according to the current mailbox view (*i.e.,* a user-defined filter). For another example, consider the `Search` request. It determines which messages in the currently selected mailbox match a list of criteria. Again, the message list that matches a search should take into account the mailbox view.

Some service variations can in fact be considered as extensions in that they open up a new family of services within the one under study. This is the case for the field view, and more specifically, views of attachments. Here, our goal is to allow attached documents to be converted into a different format to better adapt to needs and requirements. As discussed in Section 4, this particular functionality is obtained by composing the programmable IMAP server with another programmable server dedicated to remote-document processing.

After having determined the requests impacted by the service variations, we need to enable programmers to write service variations.

3. The Programmable Client-Server Model: How

Enabling service variations to be introduced in a server by a client requires addressing two main issues: (1) How to easily express a variation? (2) How to preserve the integrity of the server? Both issues are addressed in this section. Additionally, the implementation of a programmable server is discussed.

3.1. DSL Design

Issues (1) and (2) can be addressed by using a DSL approach. The idea is to design a language targeted toward specific service variations. The DSL should thus be concise and easy to use because of the dedicated nature of syntactic constructs and data types. Furthermore, programs in the DSL should be restricted enough to enable critical properties to be checked.

In the context of the IMAP case, we have designed a DSL that enables a client to define views on a mailbox. This language, called Pems, defines views at four different levels: access-point, mailbox, message, and message field.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

Access-point view. A view can be defined for a type of access-point. An access-point consists of a set of parameters such as the client terminal features, the characteristics of the link layer, and a mailbox view.

```
view accesspoint PDA {
  Mobility = YES;
  Screen   = 320 * 240;
  Color    = NO;
  Bandwidth = 10MB/s;
  Mailbox_view = nomadic(1MB);
}
```

The above example defines an access-point named `PDA`. It declares the features of the link layer and the client terminal. Also, it specifies which mailbox view to use, that is, `nomadic`. This mailbox view is invoked with an argument setting a size limit for filtering purposes. Note that parameters omitted in an access-point declaration are given a default value.

Mailbox view. This part aims to select the messages that belong to a view. A mailbox view consists of a list of pairs condition-action, sequentially executed for a given message. When a condition matches, the corresponding action is performed. An action can either drop the current message (construct **ignore**) or assign it a category of messages for its processing (construct **bind**); both actions, when executed, terminate the view and are making an implicit return. The condition-action language is inspired by Sieve [14].

To ease the programming, Pems variables are bound to the message field values, their name corresponds to a header name (*e.g.,* variable `From`), a specific message part, such as the message body (variable `Body`), or some message attribute name, such as the total message size (variable `Size`).

```
view mailbox nomadic(size s) {
  if (From == "joe@mail.fr")
    bind boss;
  if (Size > s)
    ignore;
  bind tiny;
}
```

This simple example defines a view where messages coming from a given user (*joe@mail.fr*) are systematically retained and further processed by the `boss` view. If the message size exceeds the argument value, the message is dropped. Otherwise, the message is assigned the category `tiny` and gets processed accordingly.

Message view. The idea is to define a set of fields, relevant to the client, for a given category of messages. Also, a view may be assigned to some fields to trigger specific treatments.

```
view message boss {
  From as cst("The Boss!");
  Date;
  Subject;
  Body;
  Attachment[] as bwImages(30KB);
}
```

The message view shown above consists of five fields and assigns a specific treatment to fields `From` and `Attachment[]` (via construct **as**); the other field values are reproduced verbatim. The treatment of the field `From` is defined by the view `cst`. It assigns the constant value `"The Boss!"` to the field `From`; its definition is omitted. The view `bwImages` defines the treatment of the field `Attachment[]`. It is parameterized with some size, and is applied to an attachment sequence, as indicated by the square brackets.

Field view. It aims to convert field values into some representation appropriate to the access-point. This conversion is performed by a library of functions, each taking a value, in the original format, and producing a value in a target format. These functions are local to the server and are trusted. The example shown below includes a call to the library function `blackwhite` which converts a colored image into a black-and-white one.

```
view field Attachment bwImages(size s) {
  if (Attachment.size > s)
      return "Attachment too big:" + Attachment.size;
  if (Attachment.type in "image/.*")
      return blackwhite(Attachment.value);
  ignore;
}
```

A field view is implicitly passed the value of the field. This value is accessed using the attribute `value` (*e.g.,* `Attachment.value`). Unlike other field views, the Attachment view consists of two additional attributes, namely, `size` and `type`, to access the size and the type of an attachment, respectively. Another specific aspect of the Attachment view is that it is applied to each element of an attachment sequence of a message, as illustrated above. The **return** construct is invoked to conclude the treatment of this bottom-level view.

3.2. DSL Verifications

Our approach assumes that service developers may not be trusted by the server. Furthermore, when the target family of services is related to the end-user, as in the IMAP case, the developer may not be an experienced programmer. As a consequence, the DSL should guarantee specific properties so as to both preserve the integrity of the server and prevent a faulty service to corrupt or destroy user data. Notice that, most of these requirements would not be achievable in the context of GPLs because of their unrestricted expressiveness [6]. This is mainly why untrusted (possibly buggy) services written in a GPL are usually executed on a remote machine and communicate with the server via the network. Let us examine the requirements imposed on a DSL both from the server's side and from an end-user's side.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

The Server Side

Important requirements should be fulfilled from the server's viewpoint.

Resource usage. A DSL program should use appropriate amounts of resources like CPU, memory, storage, or bandwidth. This assumes that a DSL program terminates. Although, termination is undecidable in general, the DSL can be designed so that this property be guaranteed, as illustrated by various existing DSLs (*e.g.,* Plan-P [27] and Devil [23, 22]).

In the case of Pems, programs are guaranteed to terminate because the language does not include an iteration construct; the traversing of a sequence of attachments is performed implicitly. Bandwidth could be an issue because Pems programs build messages which are then sent by the server to the client. However, these programs can only drop message fields or transform them. The latter operation only invokes trusted primitives.

Non-Interference. A DSL program should not be able to examine other users' data or modify arbitrary files on the server. Non-interference with other aspects of the server is guaranteed by an appropriate usage of resources as mentioned above.

For example, a Pems program is invoked on a specific mailbox; the only possible operations are those that select and manipulate messages.

Well-Behaved. As much as possible, the DSL should enable verifications to be performed statically, at deployment time. This is to ensure that (1) programs can be run efficiently because they require a minimum of run-time checks and (2) the server execution will not be disrupted by repetitive crashes of programs.

The End-User Side

Two main requirements are needed from the end-user's viewpoint.

Well-Behaved. Like the server, the end-user needs the DSL program to be checked statically, as much as possible. However, the reason is different: an ill-behaving program may, in general, corrupt or lose his data. Because a program cannot be checked against the programmer's intention, misbehavior may not always be detected. Nevertheless, in designing a DSL, special care can be taken to appropriately restrict the semantics of some constructs, or require the programmer to supply extra declarations at some critical places.

In designing Pems, we have paid attention to expressions selecting messages and the operations dropping fields. If misused, these aspects may overly filter out too many messages or drop too many fields.

Confidentiality. DSL programs may expose user preferences and possibly other confidential information. Consequently, the deployment process should be secure. Furthermore, it should be ensured that only the end-user (or some authorized administrator) can modify the services of an end-user. To do so, standard encryption and authentication techniques can be used.

3.3. Programmable Server Implementation

This section discusses approaches to implementing a DSL and presents a strategy to deploy services.

DSL Implementation

There are two main approaches to implement a DSL: interpretation or compilation. As discussed in Consel and Marlet [6], developing an interpreter is the easiest approach. Furthermore, it is known to be flexible, which is a key aspect when prototyping languages. However, interpretation entails a run-time overhead which may be incompatible with performance requirements of the target domain.

Traditional compilation techniques are applicable to DSLs. In fact, DSL features can even enable drastic optimizations, not possible in a GPL, and lead to better performance than equivalent programs written in a general-purpose language [9].

An hybrid approach may be used in specific cases, as advocated by Thibault *et al.* [26]. This approach relies on program specialization to remove the overhead incurred by the use of an interpreter [17]. The idea is to customize the interpreter for a given DSL program. Interestingly, specialization can occur at both compile time and run time [7]. The latter case is useful in application domains where services need to be changed frequently, calling for the use of an interpreter, at the expense of performance. This conflict between flexibility and performance can be solved by using some sort of just-in-time (JIT) compilers. Thibault *et al.* demonstrate that the effect of a JIT compiler can be achieved by specializing an interpreter at run time.

The implementation of Pems is traditional; it consists of a compiler and a run-time system. The compiler is a program generator that takes a Pems program and performs a number of verifications to fulfill the requirement on both the server and client sides, as discussed in Section 3.2. Then, it generates a C program corresponding to the implementation of the user-defined services. This code will be loaded in a server when needed, as explained below.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

This implementation strategy is chosen because it is assumed that users will not define new services frequently. Consequently, deploying new services may consist of a thorough static processing.

Service Deployment

Whether interpreted or compiled, new services can be safely and efficiently executed directly in the programmable server, provided the DSL has been suitably designed and implemented.

A remaining key issue is the binding of new services to a particular context. Indeed, the programmable server should be sensitive to such contextual aspects as the client terminal capabilities and the network features. To address this issue our approach consists of grouping these contextual aspects under the notion of access-point; a set of services is defined and tuned for a particular access-point.

Yet, we need a mechanism to bind an access-point to a particular user context so as to invoke the appropriate set of services. To do so, we use the Session Initiation Protocol (SIP) [13]. This protocol is a client-server signaling protocol, mostly known for Internet telephony. It allows a user to establish his presence and location via the so-called registrar server.

In our approach, an extra parameter is added to the registration process, to indicate the particular access-point to be considered for each programmable server. When a programmable server initiates a session with a user, it looks up its access-point at the location server. Note that this server was originally limited to storing the current location of a user.

In the context of the IMAP case, the Pems compiler produces an implementation of services for each access-point defined in a Pems program. Each implementation is registered in the programmable IMAP server under a particular access-point. To start a session, a wrapper of the IMAP client first contacts the location server to obtain the user's current access-point, in addition to enough information for the user to log into the server. It communicates this information to the programmable IMAP server so that the server can activate the appropriate set of services for the user's requests. Finally, the wrapper of the IMAP client invokes a standard IMAP client to access the mailbox. As discussed previously, programmability of the server has no impact on the client: its implementation is unchanged. In the Nova platform, we use an IMAP client named Althea that is well-suited for embedded systems.

4. Scaling Up The Programmable Client-Server Model

An important issue to cover is whether our programmable client-server model scales up. Let us address this issue by studying how programmable servers compose on a concrete case, and by presenting the families of services we tackled in the context of the Nova platform.

4.1. Composability

When developing a software system, a programmer typically delegates some treatments to specific software components. Similarly, when developing a programmable server, one would like to delegate a sub-family of services to the appropriate programmable server, if there exists one.

One instance of this situation occurred while developing the Nova platform. We independently developed a programmable version of the IMAP server and a new programmable server for remote-document processing (RDP). The latter server, and its underlying protocol, aim to enable a user to define a variety of transformations on a remote document before redirecting it to some physical or virtual output device, called a sink. Such a server is particularly useful to adapt the format of remote documents to a limited access-point. As a simple example, a user can define a service to transform a Word document into a black-and-white PDF file so as to display it on a terminal that neither runs the Word processor nor offers a color screen. Another example consists of converting a text document into an audio file which is then played on a cell phone. Notice that, we have developed this last example and actually combined RDP with yet another server to stream the audio file. This strategy avoids uploading the audio file in the cell phone and incurring some latency.

An RDP program consists of conversion rules and sinks. Conversion rules describe a graph of format conversions, whose nodes are formats (*e.g.,* PDF and Postscript) and whose edges are primitives performing conversions. Sinks are output devices, either physical (*e.g.,* a Fax machine and a printer) or virtual (*e.g.,* a Web site). Sinks are defined by a number of attributes such as the input format they require. In the example of audio files mentioned above, the streaming server is defined as a sink.

When developing the programmable version of IMAP, we realized that it could use the RDP server to process documents attached to a message, and thus, further adapt e-mail processing to the access-point. To combine both servers, a clause devoted to document processing is introduced in the Pems language. Specifically, the field view of attachments includes a clause defining conversion rules applicable to the attached documents.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

To account for this coupling of programmable servers, we modified the Pems compiler to process the new clause and to generate an RDP program. Furthermore, we added to the deployment process of a Pems program, the ability to deploy a RDP program.

4.2. The Nova Platform

We have developed a programmable platform for multimedia services, named Nova. This platform enables networking and telecommunications experts to quickly develop robust multimedia services. It consists of a programmable server and a DSL for each target application area. Four application areas are currently covered by Nova: e-mail processing (Pems), remote document processing, telephony services, and stream adapters. Let us briefly present the last two application areas, not discussed yet.

Telephony services are built upon the SIP signaling platform. We have designed a dialect of C to program call processing services. In contrast to the XML-based language called CPL, developed by Rosenberg et al. [24], our DSL is a full-fledged programming language based on familiar syntax and semantics. Yet, it conforms with the features and requirements of a call processing language as listed in the RFC 2824 [12]. In fact, our DSL goes even further because it introduces domain-specific types and constructs that allow verifications beyond the reach of both CPL and general-purpose languages. Once a session is initiated, audio communication is ensured by an appropriate tool such as the Robust Audio Tool (RAT) developed by UCL [15].

The other application area covered by Nova is stream adaptation. We have developed a language aimed to specify multimedia stream processing, named Spidle [5]. This language is used to program a server that adapts a stream to particular features of a target access-point like terminal features and the link reliability. This programmable server is a useful building block to develop other multimedia services as illustrated by the RDP server converting a text document into an audio file and piped to the streaming server to be played to the user.

The client terminals used in the Nova platform are IPaq Personal Digital Assistants with a Wi-Fi connection to an IP network. Various services have been developed for the four application areas covered by the platform.

5 Assessment

In this section, we consider the programmable version of the IMAP server to assess our approach. To do so, we studied both the server and the client side. On the client side, we assessed the benefits of some scenarios of customized accesses to a mailbox. We measured a key quantitative benefit, namely, the reduction in size of the messages fetched by the client (*e.g.,* to cope with a limited bandwidth). The choice of these scenarios is obviously subjective, therefore we tried to cover various situations that introduce different degrees of adaptation. Running these scenarios, we observed reduction factors in the server responses that range from 15 to 40. As such, these Pems programs illustrate how much can be gained by defining simple adaptation strategies, concisely expressed in a DSL.

On the server side, our assessment aims to determine the overhead of running Pems programs. To do so, we wrote Pems programs to implement the following scenarios of customized accesses to a mailbox.

`Trivial.` This scenario corresponds to a trivial identity function: it unconditionally keeps all mailbox messages. This Pems program makes it possible to measure the overhead of programmability in our IMAP server.

`Complex.` This scenario also keeps all mailbox messages but, before doing so, it performs 100 operations on each message. As such it gives us some kind of upper bound on the complexity of Pems programs.

`Misc.` We consider a set of scenarios defining various adaptation strategies and compute an average of their execution time.

We measured the total execution time of the programmable server running the above Pems programs, including loading the Pems program, selecting the mailbox and fetching all message headers in a mailbox. We compared these execution times to those of both the original IMAP server and a proxy-based programmable IMAP server. The latter server is an IMAP programmable server accessing mailboxes via requests to the original IMAP server. As such this strategy mimics a situation where programmability relies on untrusted scripts and thus requires the server to run in a separate process, or even a separate machine.

The execution times are showed in Figure 1. As can be noticed, the execution time of the **`trivial`** scenario is very close to the performance of the original IMAP server, indicating that programmability does not introduce much overhead. The average execution time of the miscellaneous scenarios (**`misc`**) is also very close to the performance of the original IMAP server. The **`complex`** scenario is about 2.5 slower than the original IMAP server. This may be seen as the worst possible slowdown considering the extensive number of operations included in **`complex`**. Finally, the proxy-based programmable server introduces an average overhead of 25%. It should noticed that this is the most favorable case. Indeed, user-defined scripts are usually run on a separate machine, especially if they are untrusted. This

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

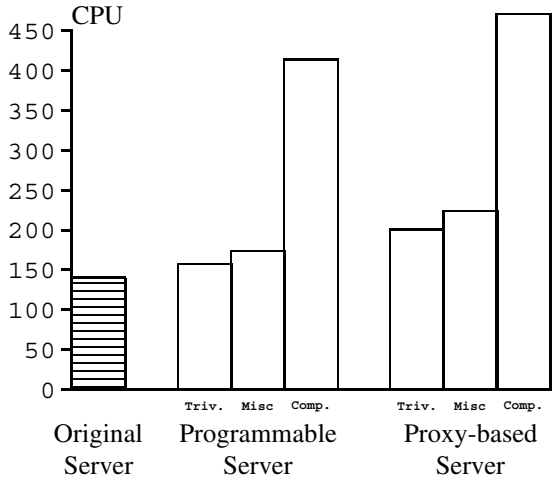

Figure 1. Execution time

separation causes a significant additional overhead because of the network latency.

6 Conclusions and Future Work

The client-server model is greatly insensitive to client needs and requirements, in that, a server behaves the same regardless of the client's terminal capabilities, network features, user preferences and evolving needs. To make the client-server model sensitive to clients, we have developed a methodology aimed to introduce programmability in a server in the form of a DSL. Our approach allows clients to program service variations in the server and to adapt it to their characteristics.

We have designed and implemented a platform, named Nova, uniformly based on programmable servers. Nova is currently composed of four programmable servers providing service programming in telephony, e-mail, remote-document processing and stream adaptation. Nova has successfully demonstrated that our approach can capture a wide spectrum of services variations without compromising robustness. Furthermore, we showed that, once made programmable, a server can adapt to a client without requiring the deployment of a new protocol or a new server implementation.

We have conducted some experiments to measure the benefits of programmable servers over existing approaches to extending servers (*e.g.,* scripts) in terms of performance and bandwidth usage. Because DSL programs do not compromise robustness, they can be executed on the same machine as the server. Furthermore, this execution requires little, if any, run-time checks. As a result, the performance of a programmable server do not introduce any significant overhead compared to its original version, as illustrated by the IMAP case.

This first step in the development of Nova has opened a number of research tracks we intend to explore. We would like to develop accurate resource usage models and analyzes supported by DSL design rules. This would be useful to introduce some admission control procedure when service variations are deployed and invoked. Resource usage would also be a useful input to some cost estimation process for service variations.

Finally, there are a number of other application areas to explore in the future, including HTTP and instant messaging. These new application areas should further refine our methodology to make a server programmable and to design a DSL to program services.

Acknowledgment.

We thank Julia Lawall from DIKU, Valérie Issarny from INRIA and the other members of the Compose group for helpful comments and discussions on earlier versions of this paper. We also thank the anonymous reviewers for their valuable inputs.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] Althea: An IMAP e-mail client for X Windows. http://althea.sourceforge.net, Feb. 2000.

[3] L. P. Barreto and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS'2002)*, Paris, France, Mar. 2002.

[4] CGI: The common gateway interface. http://cgi-spec.golux.com/ncsa.

[5] C. Consel, H. Hamdi, L. Réveillère, L. Singaravelu, H. Yu, and C. Pu. Spidle: A DSL approach to specifying streaming application. In *Second International Conference on Generative Programming and Component Engineering*, Erfurt, Germany, Sept. 2003. To appear.

[6] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Language Implementation and Logic Programming*, volume 1490 of *Lecture Notes in Computer Science*, pages 170–194, Pisa, Italy, Sept. 1998.

[7] C. Consel and F. Noël. A general approach for run-time specialization and its application to C. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 145–156, St. Petersburg Beach, FL, USA, Jan. 1996.

[8] K. Czarnecki and U. W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE

[9] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A flexible, optimizing IDL compiler. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 44–56, Las Vegas, NV, USA, June 1997.

[10] Hypertext Transfer protocol (HTTP/1.1), June 1999. Request for Comments 2616.

[11] Internet Message Access Protocol (IMAP) - version 4rev1, Dec. 1996. Request for Comments 2060.

[12] Call processing language framework and requirements, May 2000. Request for Comments 2824.

[13] Session Initiation Protocol (SIP), Mar. 2001. Request for Comments 2543.

[14] Sieve: A mail filtering language, Jan. 2001. Request for Comments 3028.

[15] RAT: Robust (unicast and multicast) Audio conferencing Tool. http://www-mice.cs.ucl.ac.uk/multimedia/software, 2002.

[16] Mozilla: an open-source web browser (version 1.3 alpha). http://www.mozilla.org, Jan. 2003.

[17] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, June 1993.

[18] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, California, Oct. 2000.

[19] Microsoft Outlook. http://www.microsoft.com/office/outlook.

[20] D. Mullet and K. Mullet. *Managing IMAP*. O'REILLY, Sept. 2000.

[21] Netscape Messenger. http://wp.netscape.com.

[22] L. Réveillère. *Approche langage au développement de pilotes de périphériques robustes*. Thèse de doctorat, Université de Rennes 1, France, dec 2001.

[23] L. Réveillère and G. Muller. Improving driver robustness: an evaluation of the Devil approach. In *The International Conference on Dependable Systems and Networks*, pages 131–140, Göteborg, Sweden, July 2001. IEEE Computer Society.

[24] J. Rosenberg, J. Lennox, and H. Schulzrinne. Programming internet telephony services. *IEEE Network Magazine*, 13(3):42–49, May 1999.

[25] Real Time Streaming Protocol (RTSP), Apr. 1998. Request for Comments 2326.

[26] S. Thibault, C. Consel, R. Marlet, G. Muller, and J. Lawall. Static and dynamic program compilation by interpreter specialization. *Higher-Order and Symbolic Computation*, 13(3):161–178, 2000.

[27] S. Thibault, C. Consel, and G. Muller. Safe and efficient active network programming. In *17th IEEE Symposium on Reliable Distributed Systems*, pages 135–143, West Lafayette, Indiana, Oct. 1998.

[28] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[29] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)
1527-1366/03 $ 17.00 © 2003 IEEE