Automatic Generation of Network Protocol Gateways

Yérom-David Bromberg[1], Laurent Réveillère[1], Julia L. Lawall[2], and Gilles Muller[3]

[1] University of Bordeaux, France
[2] University of Copenhagen, Denmark
[3] Ecole des Mines de Nantes / INRIA-Regal, France

Abstract. The emergence of networked devices in the home has made it possible to develop applications that control a variety of household functions. However, current devices communicate via a multitude of incompatible protocols, and thus gateways are needed to translate between them. Gateway construction, however, requires an intimate knowledge of the relevant protocols and a substantial understanding of low-level network programming, which can be a challenge for many application programmers.

This paper presents a generative approach to gateway construction, z2z, based on a domain-specific language for describing protocol behaviors, message structures, and the gateway logic. Z2z includes a compiler that checks essential correctness properties and produces efficient code. We have used z2z to develop a number of gateways, including SIP to RTSP, SLP to UPnP, and SMTP to SMTP via HTTP, involving a range of issues common to protocols used in the home. Our evaluation of these gateways shows that z2z enables communication between incompatible devices without increasing the overall resource usage or response time.

1 Introduction

The “home of tomorrow” is almost here, with a plethora of networked devices embedded in appliances, such as telephones, televisions, thermostats, and lamps, making it possible to develop applications that control many basic household functions. Unfortunately, however, the different functionalities of these various appliances, as well as market factors, mean that the code embedded in these devices communicates via a multitude of incompatible protocols: SIP for telephones, RTSP for televisions, X2D for thermostats, and X10 for lamps. This range of protocols drastically limits interoperability, and thus the practical benefit of home automation.

To provide interoperability, one solution would be to modify the code, to take new protocols into account. However, the code in devices is often proprietary, preventing any modification of the processing of protocol messages. Even if the source code is available, it may not be possible to install a new implementation into the device. Therefore, gateways have been used to translate between the various kinds of protocols that are used in existing appliances.

Developing a gateway, however, is challenging, requiring not only knowledge of the protocols involved, but also a substantial understanding of low-level network programming. Furthermore, there can be significant mismatches between the expressiveness of various protocols: some are binary while others are text-based, some send messages in unicast while other use multicast, some are synchronous while others are asynchronous, and a single request in one protocol may correspond to a series of requests and responses in another. Mixing this complex translation logic, which may for example involve hand coding of callback functions or continuations in the case of asynchronous responses, with equally complex networking code makes implementing a gateway by hand laborious and error prone. Enterprise Service Buses [1] have been proposed to reduce this burden by making it possible to translate messages to and from a single fixed intermediary protocol. Nevertheless, the translation logic must still be implemented by hand. Because each pair of protocols may exhibit widely differing properties, the gateway code is often not easily reusable.

This paper We propose a generative language-based approach, z2z, to simplify gateway construction. Z2z is supported by a runtime system that hides low-level details from the gateway programmer, and a compiler that checks essential correctness properties and produces efficient code. Our contributions are:

- We propose a new approach to gateway development. Our approach relies on the use of a domain-specific language (DSL) for describing protocol behaviors, message structures, and the gateway logic.
- The DSL relies on advanced compilation strategies to hide complex issues from the gateway developer such as asynchronous message responses and the management of dynamically-allocated memory, while remaining in a low-overhead C-based framework.
- We have implemented a compiler that checks essential correctness properties and automatically produces an efficient implementation of a gateway.
- We have implemented a runtime system that addresses a range of protocol requirements, such as unicast vs. multicast transmission, association of responses to previous requests, and management of sessions.
- We show the applicability of z2z by using it to automatically generate a number of gateways: between SIP and RTSP, between SLP and UPnP, and between SMTP and SMTP via HTTP. On a 200 MHz ARM9 processor, the generated gateways have a runtime memory footprint of less than 260KB, and with essentially no runtime overhead as compared to native service access.

The rest of this paper is organized as follows. Section 2 presents the range of issues that arise in implementing a gateway, as illustrated by a variety of case studies. Section 3 describes the z2z gateway architecture and introduces a DSL for describing protocol behaviors, message structures, and the gateway logic. Section 4 describes the compiler and runtime system that support this language. Section 5 demonstrates the efficiency and scalability of z2z gateways. Section 6 discusses related work. Finally, Section 7 concludes and presents future work.

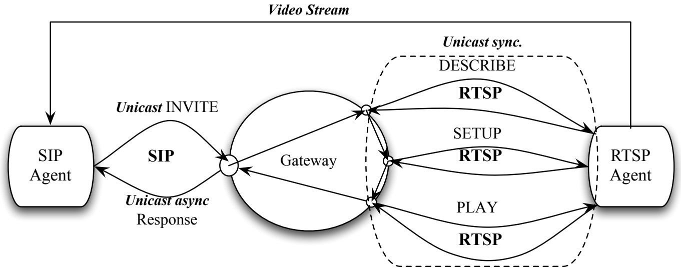


Fig. 1. SIP to RTSP gateway

2 Issues in Developing Gateways

A gateway must take into account the different degrees of expressiveness of the source and target protocols and the range of communication methods that they use. These issues are challenging to take into account individually, requiring substantial expertise in network programming, and the need to address both of them at once makes gateway development especially difficult. We illustrate these points using examples that involve a wide range of protocols.

Mismatched protocol expressiveness. The types of messages provided by a protocol are determined by the kinds of exchanges that are relevant to the targeted application domain. Thus, different protocols may provide message types that express information at different granularities. To account for such mismatches, a gateway must potentially translate a single request from the source device into multiple requests for the target device, or save information in a response from the target device for use in constructing multiple responses for the source device.

The SIP/RTSP gateway shown in Fig. 1 illustrates the case where the requests accepted by the target device are finer grained than the requests generated by the source device. This gateway has been used in the SIP-based building-automation test infrastructure at the University of Bordeaux. It allows a SIP based telephony client to be used to receive images from an Axis IP-camera.[4] This camera is a closed system that accepts only RTSP for negotiating the parameters of the video session. Once the communication is established, the gateway is no longer involved, and the video is streamed directly from the camera to the SIP client using RTP [2]. Because SIP and RTSP were introduced for different application domains, there are significant differences in the means they provide for establishing a connection. Thus, as shown in Fig. 1, for a single SIP `INVITE` message, the gateway must extract and rearrange the information available into multiple RTSP messages.

The SMTP/HTTP and HTTP/STMP gateways shown in Fig. 2 illustrate the case where information must be saved from a target response for use in constructing multiple responses for the source device. These gateways are used in a tunneling application that enables SMTP messages to be exchanged between

[4] Axis: http://www.axis.com/products/

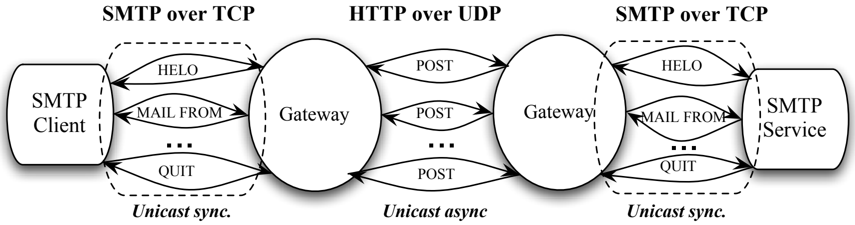


Fig. 2. HTTP tunneling gateways

two end-points over HTTP, as is useful when the port used by SMTP is closed somewhere between the source and the destination. The first gateway encapsulates an SMTP request into an HTTP message and sends it asynchronously using UDP to the second gateway, which extracts relevant information to generate the corresponding SMTP request. The response is sent back similarly.

Because all SMTP messages have to flow within the same TCP stream, the HTTP/SMTP gateway needs to know which TCP connection to use when an HTTP request is received. To address this issue, the gateway generates a unique identifier when opening the TCP connection with the destination SMTP server and includes this identifier within the HTTP response. The first gateway then includes this identifier in all subsequent HTTP requests, enabling the second gateway to retrieve the connection to use. To implement this, the first gateway needs to manage a state within a session to store the identifier returned in the first response in order to be able to find it for the subsequent requests.

Heterogeneous communication methods. Protocols differ significantly in how they interact with the network. Requests may be multicast or unicast, responses may be synchronous or asynchronous, and network communication may be managed using a range of transport protocols, most commonly TCP or UDP.

The gateway between SLP and UPnP shown in Fig. 3 involves a variety of these communication methods. Such a gateway may be used in a service discovery environment that provides mechanisms for dynamically discovering available services in a network. For example, a washing machine may search for a loudspeaker service and use it to play a sound once the washing is complete. In this scenario, the washing machine includes a SLP (Service Location Protocol) user agent and the speaker uses a UPnP (Universal Plug and Play) service agent to advertise its location and audio characteristics. UPnP is a wrapper for SSDP [3] and HTTP [4], which are used at different stages of the service discovery process.

From a multicast SLP `SrvRQST` service discovery request, the SLP/UPnP gateway extracts appropriate information, such as the service type, and sends a multicast SSDP `SEARCH` request. If a service is found, the UPnP service agent asynchronously returns a unicast SSDP response containing the URL of the service description to the gateway. Then, the gateway extracts the URL and sends a unicast HTTP `GET` request to it to retrieve the service description as an XML document. Finally, the gateway extracts information from the XML

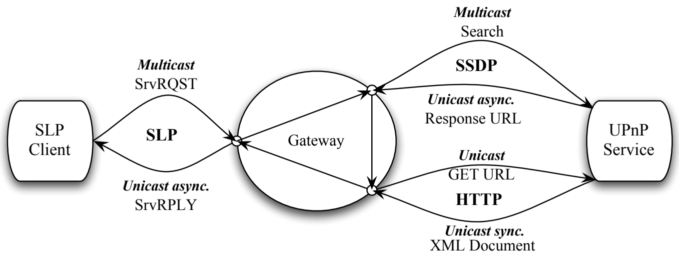


Fig. 3. SLP to UPnP gateway

document and creates an SLP `SrvRPLY` response, which it returns to the SLP client. This gateway must manage both multicast and unicast requests, and synchronous and asynchronous responses. Mixing the translation logic with these underlying protocol details complicates the development of the gateway code.

3 Specifying a Gateway Using Z2z

As illustrated in Section 2, a gateway receives a single request from the source device, translates it into one or a series of requests for one or more target devices, and then returns a response to the source device. It may additionally need to save some state when the source protocol has a notion of session that is different from that used by the target protocol, or when the interaction with the target device(s) produces some information that is needed by subsequent source requests.

Our case studies show that there are two main challenges to developing such a gateway: (1) manipulating messages and maintaining session state, to deal with the problem of mismatched protocol expressiveness, and (2) managing the interaction with the network, to deal with the problem of heterogeneous communication methods. In z2z, these are addressed through the combination of a DSL that allows the gateway developer to describe the translation between two or more protocols in a high-level way, and a runtime system that provides network interaction and data management facilities specific to the domain of gateway development. We describe the DSL in this section, and present the architecture of the runtime system in the next section.

3.1 Overview of the z2z language

To create a gateway, the developer must provide three kinds of information: 1) how each protocol interacts with the network, 2) how messages are structured, and 3) the translation logic. To allow each kind of information to be expressed in a simple way, z2z provides a specific kind of module for each of them: protocol specification (PS) modules for defining the characteristics of protocols, message specification (MS) modules for describing the structure of protocol messages, and a message translation (MT) module for defining how to translate messages

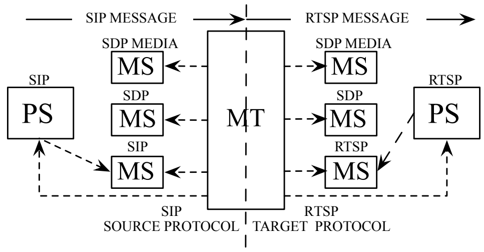


Fig. 4. The structure of a z2z gateway specification (arrows represent dependencies)

between protocols. These modules are implemented using the z2z DSL, which hides the complexities of network programming and allows specifying the relevant operations in a clear and easily verifiable way.

As illustrated in Section 2, a gateway may involve any number of protocols. Thus, a gateway specification may contain multiple protocol specification modules and message specification modules, according to the number of protocols and message types involved. A gateway specification always contains a single message translation module. Fig. 4 shows the architecture of the SIP/RTSP gateway in terms of its use of these modules. We now present these modules in more detail, using this gateway as an example.

3.2 Protocol specification module

The protocol specification module defines the properties of a protocol that a gateway should use when sending or receiving requests or responses. As illustrated in Fig. 5 for the SIP protocol, this module declares the following information.

Attributes. Protocols vary in their interaction with the network, in terms of the transport protocol used, whether requests are sent by unicast or by multicast, and whether responses are received synchronously or asynchronously. The `attributes` block of the protocol specification module indicates which combination is desired. Based on this information, the runtime system provides appropriate services. For example, SIP relies on UDP (`transport` attribute, line 2), sends requests in unicast (`mode` attribute, line 2), and receives responses asynchronously. If the transmission mode is asynchronous, the protocol specification must also include a `flow` block (line 9) describing how to match requests to responses.

Request. The entry point of a gateway is the reception of a request. On receiving a request, the gateway dispatches it to the appropriate handler in the message translation module. The `request` block (lines 3-6) of the protocol specification module declares how to map messages to handlers. For each kind of request that should be handled by the gateway, the `request` block indicates the name of the handler (`invite`, `bye`, and `ack`, for SIP), whether the request should be

```
 1  protocol sip {
 2    attributes { transport = udp/5060; mode = async/unicast; }
 3    request req {
 4      response invite when req.method == "INVITE";
 5      response bye when req.method == "BYE";
 6      void ack        when req.method == "ACK"; }
 7    sending request req { ... }
 8    sending response resp (request req) { resp.cseq = req.cseq; resp.callid = req.callid; ... }
 9    flow = { callid, cseq }
10    session_flow = { callid } }
```

Fig. 5. The PS module for the SIP protocol

acknowledged by a response (`response` if a response is allowed and `void` if no response is needed), and a predicate, typically defined in terms of the fields of the request, indicating whether a request should be sent to the given handler.

Sending. A protocol typically defines certain basic information that all messages must contain. Rather than requiring the developer to specify this information in each handler, this information is specified in a `sending` block for each of requests and responses. The `sending` block for requests is parameterized by only the request, while the `sending` block for responses is parameterized by both the corresponding request and the response, allowing elements of the response to be initialized according to information stored in the request. For example, the SIP sending block for responses copies the `cseq` and `callid` fields from the request to the response (line 8). The protocol specification module may declare local variables in which to accumulate information over the treatment of all messages.

Flow and session_flow. When a target protocol sends responses asynchronously, an incoming response must be associated with a previous request, to restart the associated handler. The `flow` block (line 9), specifies the message elements that determine this association. In SIP, a request and its matching response have the same sequence number (`cseq`) and call id (`callid`). A `session_flow` block (line 10) similarly specifies how to recognize messages associated with a session.

The information in the protocol specification module impacts operations that are typically scattered throughout the gateway. In providing the protocol specification module as a language abstraction, we have identified the elements of the protocol definition that are relevant to gateway construction and collected them into one easily understandable unit. Furthermore, creating a protocol specification module is lightweight, involving primarily selecting properties rather than implementing their support, making it easy to incorporate many different kinds of protocols into a single gateway, as illustrated by the SLP/UPnP gateway described in Section 2.

3.3 Message specification module

A network message is organized as a sequence of text lines, or of bits, for a binary protocol, containing both fixed elements and elements specific to a given message.

A gateway must extract relevant elements from the received request and use them to create one or more requests according to the target protocol(s). Similarly, it must extract relevant elements from the received responses and ultimately create a response according to the source protocol. Extracting values from a message represented as a sequence of text or binary characters is unwieldy, and creating messages is even more complex, because the element values may become available at different times, making it difficult to predict the message size and layout.

In z2z, the message specification module contains a description of the messages that can be received and created by a gateway. Based on this description, the z2z compiler generates code for accessing message elements and inserting message elements into a created message. There is one message specification module per protocol relevant to the gateway, including both the source and target protocols, as represented by the protocol specification modules, and one per any higher level message type that can be embedded in the requests and responses. For example, the SIP/RTSP camera gateway uses not only SIP and RTSP message specification modules but also message specification modules for SDP Media and SDP, which are not associated with protocol specification modules.

A message specification module provides a *message view* describing the relevant elements of incoming messages and *templates* for creating new messages. The set of elements is typically specific to the purpose of the gateway, not generic to the protocol, and thus the message specification module is separate from the protocol specification module. We illustrate the declarations of the message view and the templates in the SIP message specification module used in our camera gateway.

Message view A message view describes the information derived from received messages that is useful to the gateway. It thus represents the interface between the gateway and the message parser. Z2z does not itself provide facilities for creating message parsers, but instead makes it possible to plug in one of the many existing network message parsers[5] or to construct one by hand or using a parser generator targeting network protocols, such as Zebu [5].

Because SIP is the source protocol of the camera gateway, its message view describes the information contained in a SIP request. An excerpt of the declaration of this view is shown in Fig. 6a. It consists of a sequence of field declarations, analogous to the declaration of a C-language structure. A field declaration indicates whether the field is mandatory or optional, whether it is public or private, its type, and its name. A field is mandatory if the protocol RFC specifies that it is always present, and optional otherwise. A field is public if it can be read by the gateway logic, and private if it can only be read by the protocol specification. The type of a field is either integer, fragment, or a list of one of these types. A field of type fragment is represented as a string, but the gateway logic can cause it to be parsed as a message of another protocol, such as SDP or SDP Media, in our example.

[5] oSIP: http://www.gnu.org/software/osip
Sofia-SIP: http://opensource.nokia.com/projects/sofia-sip/
Livemedia: http://www.livemediacast.net/about/library.cfm

```
read {
  mandatory private int cseq;
  mandatory private fragment callid;
  mandatory private fragment via;
  mandatory private fragment to;
  mandatory private fragment from;
  mandatory private fragment method;

  optional private fragment to_tag;
  optional private int cseqsss;

  mandatory public fragment uri, body;
  mandatory public fragment from_host;
}
```

a) View of SIP requests

```
 1 response template Invite_ok {
 2   magic = "foo";
 3   newline = "\r\n";
 4   private fragment from, to, callid, via, contact;
 5   private int cseq, content_length;
 6   public fragment body, to_tag;
 7 --foo
 8 SIP/2.0 200 OK
 9 Via: <%via%>
10 [...]
11 Content-Length: <%content_length%>
12
13 <%body%>
14 --foo }
```

b) Template for an INVITE method success response

Fig. 6. SIP message specification for the camera gateway

Templates Z2z maintains messages to be created as a pair of a template view and a template. The template language is adapted from that of Repleo [6]. A message is created in the message translation module by making a new copy of the template view, and initializing its fields, in any order. At a send or return operation in the message translation module, the template representing the message is flushed, filling its holes with the corresponding values from the view.

Because SIP is the source protocol of our camera gateway, its templates describe the information needed to create SIP responses. Typically, there are multiple response templates for each method, with one template for each relevant success and failure condition. Fig. 6b shows the template for a response indicating the success of an `INVITE` request.

A template declaration has three parts: the structural declarations (lines 2-3), the template view (lines 4-6), and the template text (lines 7-14). The structural declarations indicate a string, `magic`, marking the start and end of the template text, and the line separator, `newline`, specified by the protocol RFC. The template view is analogous to the message view, except that the keywords `mandatory` and `optional` are omitted, as all fields are mandatory to create a message. The private fields are filled in by the `sending` block of the protocol specification. The public fields are filled in by the message translation module. Finally, the template text has the form of a message as specified by the protocol RFC, with holes delimited by `<%` and `%>`. These holes refer to the fields of the template view, and are instantiated with the values of these fields when the template is flushed. Binary templates, as needed for SLP messages in our service discovery gateway (Section 2), can be defined, using the keyword `binary`.

3.4 Message translation module

The message translation module expresses the message translation logic, which is the heart of the gateway. This module consists of a set of handlers, one for each kind of relevant incoming request, as indicated by the protocol specification module. Handlers are written using a C-like notation augmented with domain-specific

```
1  fragment session_id = "";
2
3  sip response invite (sip request s) {
4    rtsp response rr;
5    sip  response sr, failed;
6    sdp_media message rtsp_m, sip_m, media_resp;
7    sdp message sdp_rtsp, sdp_sip, sdp_resp;
8    fragment list inv_medias, rtsp_medias;
9
10   // Create error response
11   failed=Invite_failure(code=400,to_tag=random());
12
13   sdp_rtsp = (sdp message)(s.body);
14   inv_medias = (fragment list)(sdp_rtsp.medias);
15
16   // Notify that something is happening
17   preturn Invite_provisional(body = "",
18                   to_tag = random());
19
20   // Retrieve the description of a media object
21   rr = send(Describe(resource = s.uri_uname));
22   if (empty(rr.body)) return failed;
23   sdp_sip = (sdp message)(rr.body);
24   rtsp_medias = (fragment list)(sdp_sip.medias);
25
26   // See whether a compatible video format exists
27   foreach (fragment rtsp_m_ = rtsp_medias) {
28     rtsp_m = (sdp_media message)rtsp_m_;
29     if (rtsp_m.type == "video") {
30       foreach (fragment sip_m_ = inv_medias) {
31         sip_m = (sdp_media message)sip_m_;
32 if ((rtsp_m.type == sip_m.type) &&
33     (rtsp_m.profile == sip_m.profile)) {
34   // Found something compatible
35   if (empty(rtsp_m.control))
36     return failed;
37   // Specify the transport mechanism
38   rr = send(Setup(uri=rtsp_m.control,
39                destination=s.from_host,
40                port1=sip_m.port,
41                port2=sip_m.port+1));
42   if (empty(rr.sessionId) ||
43       empty(rr.code) || rr.code > 299)
44     return failed;
45   session_start();
46   session_id = rr.sessionId;
47   // Tell the server to start sending data
48   rr = send(Play(resource = s.uri_uname,
49                sessionId = session_id));
50   if (empty(rr.code) || rr.code > 299) {
51     session_end(); return failed; }
52   media_resp = Media(type = sip_m.type,
53                  profile = sip_m.profile);
54   if (empty(rr.server_port))
55     media_resp.port = 0;
56   else media_resp.port = rr.server_port;
57   sdp_resp = Sdp_media(header=
58         sdp_rtsp.header,media=media_resp);
59   return Invite_ok(body = sdp_resp,
60                  to_tag = random());
61 }}}}
62 return failed; }
```

Fig. 7. The INVITE handler of the message translation module for the camera gateway

operators for manipulating and constructing messages, for sending requests and returning responses, and for session management. Fig. 7 shows the invite handler for the camera gateway.

Manipulating message data A handler is parameterized by a view of the corresponding request. The information in the view can be extracted using the standard structure field access notation (line 13). If a view element is designated as being optional in the message specification module, it must be tested using empty to determine whether its value is available before it is used (line 22). A view element of type fragment can be cast to a message type, using the usual type cast notation. In line 23, for example, the body of the request is cast to an SDP message, which is then manipulated according to its view (line 24).

A handler creates a message by invoking the name of the corresponding template (line 17). Keyword arguments can be used to initialize the various fields (lines 17-18) or the fields can be filled in incrementally (lines 54-56). A created message is maintained as a view during the execution of the handler and flushed to a network message at the point of a send or return operation.

Sending requests and returning responses A request is sent using the operator send, as illustrated in line 38. If the protocol specification module for the corresponding target protocol indicates that a response is expected, then execution pauses until a response is received, and the response is the result of the send operation. If the protocol specification indicates that no response is expected, send returns immediately. There is no need for the developer to break the handler up into a collection of callback functions to receive asynchronous responses, as is required in most other languages used for gateway programming. Instead, the difference between synchronous and asynchronous responses is handled by the z2z compiler, as described in Section 4. This strategy makes it easy to handle the case where the gateway must translate a single request from the source device into multiple requests for the target device, requiring multiple send operations.

If the protocol specification module indicates a return type for a handler, then the handler may return a response. This is done using return (line 59), which takes as argument a message and terminates execution of the handler. A provisional response, as is needed in SIP to notify the source device that a message is being treated, can be returned using preturn (line 17). This operator asynchronously returns the specified message, and handler execution continues.

Session management A session is a state that is maintained over a series of messages. If the protocol specification module for the source protocol declares how messages should be mapped to sessions (session_flow), then the message translation module may declare variables associated with a session outside of any handler. The camera gateway, for example, declares the session variable session_id in line 1. The message translation module initiates a session using session_start() (line 45). Once the session has started any modification made to these variables persists across requests within the session, until the session is ended using session_end(). At this point, all session memory is freed. The SMTP/HTTP/SMTP gateways described in Section 2 similarly use sessions to maintain the TCP connection identifier across multiple requests.

4 Implementation

Our implementation of the z2z gateway generator comprises a compiler for the z2z language and a runtime system. From the z2z specification of a gateway, the z2z compiler generates C code that can then be compiled using a standard C compiler and linked with the runtime system. The generated code is portable enough to run on devices ranging from desktop computers to constrained devices such as PDAs or home appliances. The runtime system defines various utility functions and amounts to about 7500 lines of C code. The z2z compiler is around 10500 of OCaml code. Note that the compiler can be used offline to produce the gateway code and therefore is not required to be present on the gateway device. We first describe the verifications performed by the compiler, then present the main challenges in code generation, and finally present the runtime system.

4.1 Verifications

The z2z compiler performs consistency checks and dataflow analyses to detect erroneous specifications and to ensure the generation of safe gateway code.

Consistency checks. As was shown in Fig. 4, there are various dependencies between the modules making up a z2z gateway. The z2z compiler performs a number of consistency checks to ensure that the information declared in one module is used elsewhere according to its declaration. The main inter-module dependencies are derived from the `request` and `sending` blocks of the protocol specification and the types and visibilities of the elements of the message views. The `request` block associated with the source protocol declares how to dispatch incoming requests to the appropriate handlers and whether a response is expected from these handlers. The z2z compiler checks that the message translation module defines a handler for each kind of message that should be handled by the gateway and that each handler has an appropriate return type. The `sending` block of a protocol specification module initializes some fields for all requests or responses sent using that protocol. The compiler checks that every template view defined in the corresponding message specification module includes all of these fields. Finally, the message specification module indicates for each field of a view the type of value that the field can contain and whether the field can be accessed by the message translation module (`public`) or only by the protocol specification module (`private`). The z2z compiler checks that the fields are only used in the allowed module and that every access or update has the declared type.

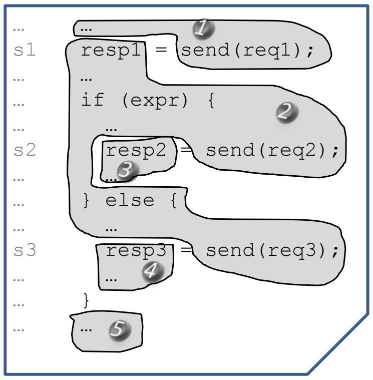
Dataflow analyses. The z2z compiler performs a dataflow analysis within the message translation module to ensure that values are well-defined when they are used. The principal issues are in the use of optional message fields, session variables, and created messages. A message specification module may declare some message fields as `optional`, indicating that they may be uninitialized. The z2z compiler enforces that any reference to such a field is preceded by an `empty` check. Session variables cannot be used before a `session_start` operation or after a `session_end` operation. The z2z compiler checks that references to these variables do not occur outside these boundaries. Finally, the z2z compiler checks that all public fields of a template are initialized before the template is passed to `send` and that all execution paths through the `sending` block of the corresponding protocol specification module initialize all `private` fields.

4.2 Code generation

The main challenges in generating code from a z2z specification are the implementation of the `send` operation, the implementation of the variables used by the message translation module, and the implementation of memory management.

The `send` *operation.* The handlers of a z2z message translation module are specified as sequential functions, with `send` having the syntax of a function call that may return a value. If the target protocol returns responses synchronously, the z2z compiler does indeed implement `send` as an ordinary function call. If the target protocol returns responses asynchronously, however, this treatment is not sufficient. In this case, the implementation of `send` does not return a value, but must instead receive as an argument information about the rest of the handler so that the handler can be restarted when a response becomes available. The standard solution is to decompose the code into a collection of callback functions, which are tedious, error-prone, and unintuitive to write by hand. Fortunately, it has been observed that such callback functions amount to *continuations*, which can be created systematically [7]. The z2z compiler thus splits each handler at the point of each `send` to create a collection of functions, of which the first represents the entry point of the handler and the rest represent some continuation.

Fig. 8 illustrates the splitting of a handler performed by the z2z compiler when the target protocol of a `send` returns responses asynchronously. This code contains three `send` operations, one on line s1 and the others in each of the if branches (lines s2 and s3). The continuation function for the `send` on line s1 contains the code in the region labeled (2). The continuation function for the `send` on line s2 contains the code in region (3) and the one for the `send` on line s3 contains the code in region (4). As shown, the latter two continuation functions explicitly contain only the code within the corresponding if branch. The execution of the handler must, however, continue to the code following the if statement, which is in the continuation of both `send` operations. To reduce the code size, the z2z compiler factorizes the code after the if statement into a separate continuation function, labeled (5), which is invoked by both (3) and (4) after executing the if branch code [8].



```
...   ...
s1    resp1 = send(req1);
...   ...
...   if (expr) {
...     ...
s2      resp2 = send(req2);
...     ...
...   } else {
...     ...
s3      resp3 = send(req3);
...     ...
...   }
...   ...
```

Fig. 8. Code slicing for continuations

Variables. Splitting a handler into a set of disjoint continuation functions in the asynchronous case complicates the implementation of the handler's local variables when these variables are used across `send`s and thus by multiple continuations. The z2z compiler identifies handler variables whose values must be maintained across asynchronous `send`s, and implements them as elements of an environment structure that such a `send` passes to the runtime system. The runtime system stores this environment, and passes it back to the stored continuation function when the corresponding response is received.

Session variables are similarly always implemented in an environment structure, as by design they must be maintained across multiple invocations of the message translation module. Between handler invocations, the runtime system stores this environment with the other information about the session.

Dynamic memory management. Template constructors dynamically allocate memory, analogous to Java's `new` operation. This memory may be referenced from arbitrary local and session variables of the message translation module, and must be freed when no longer useful. Had we translated z2z into a garbage collected language such as Java, then we could rely on the garbage collector to free this memory. We have, however, used C, to avoid the overhead of including a fully-featured runtime system such as the JVM. To avoid the risk of dangling pointers, the z2z compiler generates code to manage reference counts [9]. An alternative, for future work, is to use a garbage collector for C code [10].

4.3 Runtime system

The z2z runtime system implements a network server capable of simultaneously handling many messages, that may rely on various protocols. The server is parameterized by the information specific to each protocol, as provided in the corresponding protocol specification module (Section 3.2). When a message is received, the runtime system invokes the corresponding parser to construct a message view as defined in the message specification module. The runtime system then executes the code generated from the message translation module for the handler corresponding to the incoming request. The runtime system also provides various utility functions that are used by the code generated by the z2z compiler to send requests synchronously or asynchronously, to save and restore environments, to manage sessions, and to perform various other operations.

Receiving network messages The z2z runtime system is designed to efficiently juggle many incoming requests simultaneously. It is multithreaded, based on the use of a single main thread and a pool of worker threads. The main thread detects an incoming connection, and then assigns the processing of this connection to an available worker thread. The pool of worker threads avoids the high overhead that would be entailed by spawning a new thread per connection, and thus contributes to the overall efficiency of the approach. It furthermore avoids the mutex contention that would be incurred by the use of global shared variables. The z2z developer does not need to be aware of these details.

TCP poses further challenges. In this case, a stream of messages arrives within a single connection. Depending on the protocol, substantial computation may be required to isolate the individual messages within a stream. To avoid dropping messages, the main thread assigns two worker threads to a TCP connection. One, the *producer*, receives data from the incoming TCP stream and separates it into messages, while the other, the *consumer*, applies the gateway logic. These threads communicate via shared memory. When the producer has extracted a message from the incoming stream, it sends a signal to the consumer, which then reads the message in the shared memory and processes it. On completion, it sends a signal to the producer. This approach allows the main thread to multiplex I/O on a set of server sockets to provide gateway service to multiple devices.

Network protocol gateways furthermore must manage multiple concurrent connections between many devices. This adds significantly to the complexity of the gateway implementation. For example, to communicate with a device that uses SMTP, multiple requests must be sent inside the same TCP connection, while for a RTSP device each request requires the creation of a new TCP connection. The z2z runtime system hides these details from the gateway developer. As illustrated by the HTTP/SMTP example, the runtime system must keep open the TCP connection used to send SMTP requests even if incoming requests are responded to asynchronously. However, in this case, subsequent incoming requests are not related to each other and the runtime system needs to know which TCP connection to use. To address this issue, the runtime system maintains a table of current active TCP connections and provides references to them, so that they can be retrieved later. The runtime system can also seamlessly switch from IPv4 to IPv6, send messages in unicast or multicast, and use UDP or one or many TCP connections, as specified by attributes in the protocol specification module, without requiring any additional programming from the gateway developer.

Processing a message When a thread is assigned the processing of a message, it executes the message parser of the corresponding protocol to construct a message view, as described in Section 3.3. Then, it calls the `dispatch` function, generated by the z2z compiler from the PS module, to select the handler to execute. If a handler sends requests asynchronously, the runtime system explicitly suspends the control flow and saves the current continuation, handler state, and session state in a global shared memory. The local memory allocated for the current thread is freed and the thread returns to the main pool. When a response is received by the main thread, the runtime system assigns its processing to an available worker thread, restores the corresponding states and continuation, and execution of the handler continues.

5 Evaluation

To assess our approach, we have implemented the SIP/RTSP, SLP/UPnP, SMTP/HTTP and HTTP/SMTP gateways described in the case studies of Section 2. In the latter case, although our experiments use HTTP over UDP or TCP for the encapsulation, it is possible to use other protocols such as SIP over UDP. We have implemented our gateways on a Single Board Computer (SBC) to represent the kind of limited but inexpensive or energy-efficient devices that are found in PDAs, mobile devices and home appliances. We use a Eukréa CPUAT91 card,[6] based on a 200 MHz ARM9 processor. The SBC has 32MB of SDRAM, 8MB of flash memory, an Ethernet controller, and runs a minimal Linux 2.6.20 kernel. For the SIP/RTSP experiment, we use the open-source Linphone video-phone client[7] and an Axis RTSP camera. For the SLP/UPnP experiment, we use a handcrafted SLP client based on the INDISS framework [11] and a UPnP service provided by CyberLink.[8] For the SMTP/HTTP/SMTP experiment, we use the multi-threaded SMTP test client and server distributed with Postfix [12] to stress the generated gateways.

Fig. 9 shows the size of the various specifications that must be provided to generate each gateway. A protocol specification module is independent of the

[6] Eukréa. http://www.eukrea.com/

[7] Linphone: http://www.linphone.org/

[8] CyberLink: http://www.cybergarage.org/net/upnp/java/

			Input specification			Parser wrapper	Z2z gateway (size in KB)		
			(lines of z2z code)			(lines of C code)	Generated	Runtime	Total
			PS	MS	MT	Parser or wrapper	modules	system	
SIP/RTSP		SIP	24	118	102	168	72	80	152
		RTSP	20	104		210			
		SDP	-	12		52			
		SDP_media	-	15		83			
SLP/UPnP		SLP	12	21	5	166	44	80	124
		SSDP	6	31		223			
		HTTP	9	43		178			
SMTP/HTTP	UDP	SMTP	10	23	83	96	40	80	120
		HTTP	9	92		103			
	TCP alive	SMTP	10	23	71	96	36		116
		HTTP	9	64		105			
	TCP non-alive	SMTP	10	23	83	96	36		
		HTTP	9	92		114			
HTTP/SMTP	UDP	HTTP	9	43	69	160	32	80	112
		SMTP	10	23		44			
	TCP alive	HTTP	9	43	63	488	36		116
		SMTP	10	23		34			
	TCP non-alive	HTTP	9	43	75	190	32		112
		SMTP	10	23		44			

Fig. 9. The size of the input specifications and the generated gateway

Native service access

	SIP↔SIP	RTSP↔RTSP
Time	351	701

	SLP↔SLP	UPnP↔UPnP
Time	2	58

Z2z

	SIP/RTSP	SLP/UPnP
Time	986	78

(a)

Native service access - SMTP/SMTP

Nb client	Mail size (KB)	Nb Mail	Time
1	1	1	10
1	10	1	10
1	10	10	15
10	10	10	15

Z2z - SMTP/HTTP/SMTP

	Nb client	Mail size (KB)	Nb Mail	Time	increase factor
UDP	1	1	1	50	5
	1	10	1	65	6.5
	1	10	10	645	43
	10	10	10	146	9.7
TCP alive	1	1	1	48	4.8
	1	10	1	50	5
	1	10	10	410	27.3
	10	10	10	98	6.5
TCP non-alive	1	1	1	53	5.3
	1	10	1	53	5.3
	1	10	10	400	26.6
	10	10	10	111	7.4

(b)

Fig. 10. Native service access vs. z2z (ms)

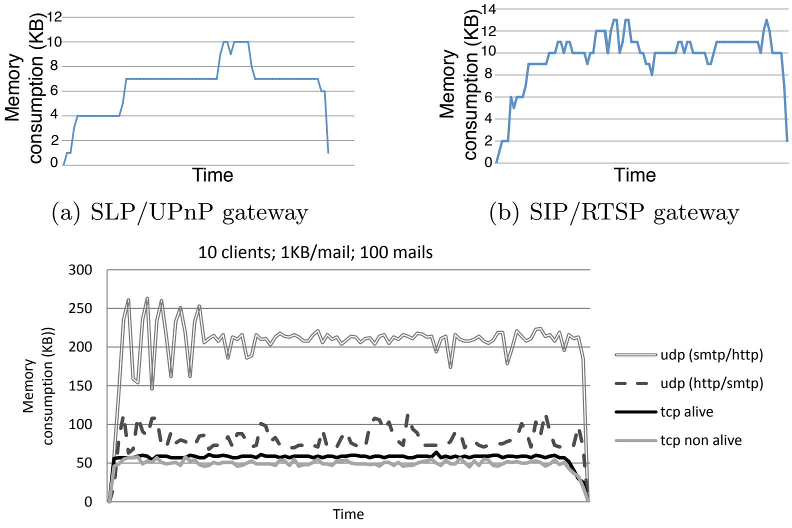
targeted gateway and thus can be shared by all gateways relevant to the protocol. The message translation module for the SLP/UPnP gateway is particularly simple, being only 5 lines of code. Indeed, the complete z2z specification for this gateway is less than 100 lines of code. As described in Section 3.3, the message specification module must provide a parser for incoming messages. For our experiments, we have implemented simple parsers for each message type, amounting to at most 488 lines of C code. Each of the generated gateways does not exceed, in the worst case, 150KB of compiled C code, including 80KB for the runtime system.

Fig. 10 shows the response time for the SIP/RTSP, SLP/UPnP, and SMTP/-HTTP/SMTP gateways, as well as the native protocol communication costs. In each case, at the client side we measure the time from sending an initial request to receiving a successful response. These experiments were performed using the loopback interface to remove network latency. As illustrated in Fig. 10(a), the z2z implementation of the SIP/RTSP gateway does not introduce any overhead as compared to an ideal case of zero-cost message translation, since its response time is less than the sum of the response times for SIP and RTSP separately. The response time for the SLP/UPnP gateway is a little more than that of SLP and UPnP separately. The cost of the z2z gateway is due in part to the polling done by the main thread in the case of asynchronous responses, as described in Section 4.3. This strategy introduces some time overhead, but reduces memory requirements. The response time for UPnP, furthermore, depends heavily on the stack that is used. If we use a Siemens stack[9] rather than the CyberLink stack, the native UPnP time rises to 593ms, substantially higher than the gateway cost.

Fig. 10(b) shows the performance of the gateways generated in order to tunnel SMTP traffic into HTTP. We consider three types of tunnel according to the nature of transport layer protocol being used to exchange either asynchronously (i.e. UDP or $\text{TCP}_{\text{non-alive}}$) or synchronously (i.e. $\text{TCP}_{\text{alive}}$) messages between the two tunnel end-points. When sending one e-mail, with a size from 1KB up to 10 KB, passing through a HTTP tunnel roughly increases the native response time by a factor of 5 whatever the tunnel considered. This overhead comes primarily from the mailing process. According to the SMTP RFC5321, sending an e-mail involves sending at least 5 SMTP commands and acknowledgements, of at most 512 bytes each, and splitting the mail content in order to send chunks of at most 998 bytes. Commands, acknowledgements, and data chunks are packets that need to be encapsulated into the HTTP protocol and de-encapsulated on both sides of the HTTP tunnel. Increasing the number of packets being encapsulated at tunnel end-points inherently increases the response time. Our experimental results show that sending a sequence of 10 emails increases the response times at most by 43 as compared to a native implementation in the worst case. However, when the mail is sent in parallel (10 clients), response times only increase by a factor of at most 10. The latter result highlights the efficiency of the parallelism provided by our generated gateways. Furthermore, note that the SMTP test server simply throws away without processing the messages received from network, whereas the tunnel end-points must process them, therefore increasing the response time. The native response time obtained with a real deployed SMTP server such as Postfix [12] may be up to 18x slower than the SMTP test server, which is much closer to the response times obtained via the generated gateways.

Finally, Fig. 11 shows the amount of dynamic memory used during the lifetime of each of our generated gateways. The memory footprint is directly related to the network input/output traffic. When idle, the memory consumption is low and does not exceed 2KB for the considered gateways. At peak time, both the SIP/-RTSP and SLP/UPnP gateways consume at most 14KB, as shown in Figs. 11(a) and 11(b). Comparatively, the SMTP/HTTP/SMTP gateways based on TCP use only at most 60KB to process 100 1KB mails sent by 10 simultaneous clients. However, the memory consumption of the UDP-based gateway may reach up to 159KB-260KB as shown in Fig. 11(c). This overhead is inherent in the use of UDP, as a dedicated 1500 byte buffer is allocated for each incoming UDP packet. TCP, on the other hand, enables reading variable-size messages into a stream.

[9] Siemens UPnP: http://www.plug-n-play-technologies.com/



(a) SLP/UPnP gateway

(b) SIP/RTSP gateway

(c) HTTP tunnelling

Fig. 11. Dynamic memory consumption (KB)

6 Related Work

Other approaches to interoperability. The middlewares ReMMoC [13], RUNES [14], MUSDAC [15], and BASE [16] for use in networked devices allow the device code to be developed independently of the underlying protocol. Plug-ins then select the most appropriate communication protocol according to the context. Many applications, however, have not been developed using such middleware systems and cannot be modified because their source code is not available.

Bridges provide interoperability without code modification. Direct bridges, such as RMI-IIOP[10] and IIOP-.NET,[11] provide interoperability between two fixed protocols. A direct bridge must thus be developed separately for every pair of protocols between which interaction is needed. The diversity of protocols that are used in a networked home implies that this is a substantial development task. Indirect bridges such as Enterprise Service Buses (ESBs) [1], translate messages to and from a single fixed intermediary protocol. This approach reduces the development effort, but may limit expressiveness, as some aspects of the relevant protocols may not be compatible with the chosen intermediary protocol. INDISS [11] and NEMESYS [17] also use a single intermediary protocol, but one that is specific to the protocols between which interoperability is desired. Still, none of these approaches addresses the problem of implementing the bridge, which requires a thorough knowledge of the protocols involved and low-level

[10] RMI-IIOP:http://java.sun.com/products/rmi-iiop/

[11] IIOP-.NET: http://iiop-net.sourceforge.net/index.html

network programming. This makes it difficult to quickly integrate devices that use an unsupported protocol into a home environment.

Z2z can be used in the context of either direct or indirect bridges. Our approach targets the weak point of both: the difficulty of bridge development. We propose a high-level interface definition language that abstracts away from network details, makes relevant protocol properties explicit, provides static verification at the specification level, and automatically generates low-level code.

Compilation. Z2z uses a number of advanced compilation techniques to be able to provide a high-level notation while still generating safe and efficient code. Krishnamurthi *et al.* [7] pioneered the use of continuations to overcome the asynchrony common in web programming. Our implementation of continuations in C code is based on that presented by Friedman *et al.* [18]. Our dataflow analysis uses standard techniques [19], adapted to the operations of the z2z DSL. Finally, reference counting has long been used to replace garbage collection [9].

7 Conclusion and Future Work

In this paper, we have presented a generative language-based approach, z2z to simplify gateway construction, a problem that has not been considered by previous frameworks for gateway development. Z2z is supported by a runtime system that hides low-level networking intricacies and a compiler that checks essential correctness properties and produces efficient code. We have used z2z to automatically generate gateways between SIP and RTSP, between SLP and UPnP, and between SMTP and SMTP via HTTP. The gateway specifications are 100-400 lines of z2z code while the generated gateways are at most 150KB of compiled C code and run with a runtime memory footprint of less than 260KB, with essentially no runtime overhead.

We are currently extending the z2z approach to generate code that can be deployed on existing middleware systems such as ReMMoC [13]. We are also exploring the extension of z2z to enable dynamic adaptation of gateway code according to context information. Following our approach, there are a number of other application areas to explore in the future, including Web services and network supervision. These new application areas should enable us to further refine our language, compiler and runtime system, making z2z a practical approach for gateway construction. Finally, we are considering how z2z can efficiently handle failures within the participants of an interaction. To address this issue, we are defining language extensions to specify failure recovery policies and runtime primitives to support these new features.

Availability Source code: http://www.labri.fr/perso/reveille/projects/z2z/

References

1. Chappell, D.: Enterprise Service Bus. O'Reilly (2004)

2. Perkins, C.: RTP - Audio and Video for the Internet. Addison-Wesley (2003)
3. Goland, Y.Y., Cai, T., Leach, P., Gu, Y.: Simple service discovery protocol/1.0: Operating without an arbiter. http://quimby.gnus.org/internet-drafts/draft-cai-ssdp-v1-03.txt (October 1999)
4. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard) (June 1999) Updated by RFC 2817.
5. Burgy, L., Réveillère, L., Lawall, J.L., Muller, G.: A language-based approach for improving the robustness of network application protocol implementations. In: 26th IEEE International Symposium on Reliable Distributed Systems, Beijing (October 2007) 149–158
6. Arnoldus, J., Bijpost, J., van den Brand, M.: Repleo: a syntax-safe template engine. In: GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering, NY, USA, ACM (2007) 25–32
7. Krishnamurthi, S., Hopkins, P.W., McCarthy, J., Graunke, P.T., Pettyjohn, G., Felleisen, M.: Implementation and use of the PLT Scheme web server. Higher-Order and Symbolic Computation **20**(4) (2007) 431–460
8. Steele Jr., G.L.: Lambda, the ultimate declarative. AI Memo 379, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts (November 1976)
9. Cohen, J.: Garbage collection of linked data structures. ACM Computing Surveys **13**(3) (September 1981) 341–367
10. Boehm, H., Weiser, M.: Garbage collection in an uncooperative environment. Software Practice & Experience **18**(9) (September 1988) 807–820
11. Bromberg, Y.D., Issarny, V.: INDISS: Interoperable discovery system for networked services. In: Proceedings of the 6th International Middleware Conference, Grenoble, France (November 2005) 164–183
12. Hildebrandt, R., Koetter, P.: The book of Postfix: state-of-the-art message transport. NO-STARCH (2005) http://www.postfix.org/.
13. Grace, P., Blair, G.S., Samuel, S.: A reflective framework for discovery and interaction in heterogeneous mobile environments. SIGMOBILE Mob. Comput. Commun. Rev. **9**(1) (2005) 2–14
14. Costa, P., Coulson, G., Mascolo, C., Mottola, L., Picco, G.P., Zachariadis, S.: Reconfigurable component-based middleware for networked embedded systems. In International Journal of Wireless Information Networks **14**(2) (June 2007) 149–162
15. Raverdy, P.G., Issarny, V., Chibout, R., de La Chapelle, A.: A multi-protocol approach to service discovery and access in pervasive environments. In: The 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services, San Jose, CA, USA (July 2006) 1–9
16. Becker, C., Schiele, G., Gubbels, H., Rothermel, K.: Base: A micro-broker-based middleware for pervasive computing. In: PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications, Washington, DC, USA, IEEE Computer Society (2003) 443
17. Bromberg, Y.D.: Solutions to middleware heterogeneity in open networked environment. Phd Thesis, INRIA/UVSQ (2006)
18. Friedman, D.P., Wand, M., Haynes, C.T.: Essentials of Programming Languages. MIT Press (1992)
19. Appel, A.: Modern Compiler Implementation in ML. Cambridge University Press (1998)