EZ: Towards Efficient Asynchronous Protocol Gateway Construction

Yérom-David Bromberg[1], Floréal Morandat[1],
Laurent Réveillère[1], and Gaël Thomas[2]

[1] LaBRI University of Bordeaux, France
[2] LIP6-INRIA University of Pierre et Marie Curie

Abstract. Over the past decade, we have witnessed the emergence of a bulk set of devices, from very different application domains interconnected *via* Internet to form what is commonly named Internet of Things (IoT). The IoT vision is grounded in the belief that all devices are able to interact seamlessly with each other anytime, anyplace, anywhere. However, devices communicate *via* a multitude of incompatible protocols, and consequently drastically slow down the IoT vision adoption. Gateways, that are able to translate one protocol to another, appear to be a key enabler of the future of IoT but present a cumbersome challenge for many developers. In this paper, we are providing a framework called **EZ** that enables to generate gateways for either C or Java platform without requiring from developers any substantial understanding of either relevant protocols or low-level network programming.

1 Introduction

In the new era of the Internet Of Thing (IoT), interoperability is a key challenge. Over the last years, a promising solution that gained success to address interoperability issues is to use gateways that translate back and forth messages among heterogeneous protocols [1–5]. The design of such gateways [1–5] does not take as a first class priority the specific needs of the IoT context. First, gateways must scale in the large, i.e., they must efficiently manage both bulk sets of messages and simultaneous message translation processes. Second, gateways need to be as pervasive as possible: they must run on highly heterogeneous software environments. Current gateways only target low-level C code and are not adequate in the IoT context where Java is also a mainstream language.

In this paper, we propose `EZ` a new gateway compiler for the `z2z` language [2] that relies on the event paradigm. We choose the `z2z` language as it has already proven to be adequate to describe gateways. To solve performance issues `EZ` defines a workflow of handlers, which communicate with events. Further, to take into account software heterogeneity, `EZ` is able to compile the `z2z` language to both C and Java code, ready to be plugged into respective runtimes.

The remainder of this paper is structured as follows. Section 2 introduces our approach to generate from high level specifications our next generation gateways for either C or Java environment. Section 2.2 is focused on the internal design of

J. Dowling and F. Taïani (Eds.): DAIS 2013, LNCS 7891, pp. 169–174, 2013.

our next generation gateways. In particular, it presents our new asynchronous runtime system for building scalable gateways in either C or Java along with EZ. Section 3 presents the performance evaluation of our asynchronous gateways. Finally, Section 4 reviews related research works and Section 5 concludes the paper with a discussion of future research directions.

2 EZ Approach

In a way similar to z2z, zebu [2,6], our approach is based on generative programming. More specifically, EZ reuses the z2z domain specific language to specify generated gateways (Figure 1, ❶). However, EZ introduces a new compiler enabling the generation of asynchronous code in either C or Java to be linked into an adequate and efficient event-based oriented runtime system (Figure 1, ❷, ❸) that fulfills the requirements of IoT. In other terms, developers only have to replace the z2z compiler by the EZ one to generate event-based gateways. These gateways use exclusively non-blocking system calls to perform asynchronous I/O network operations.

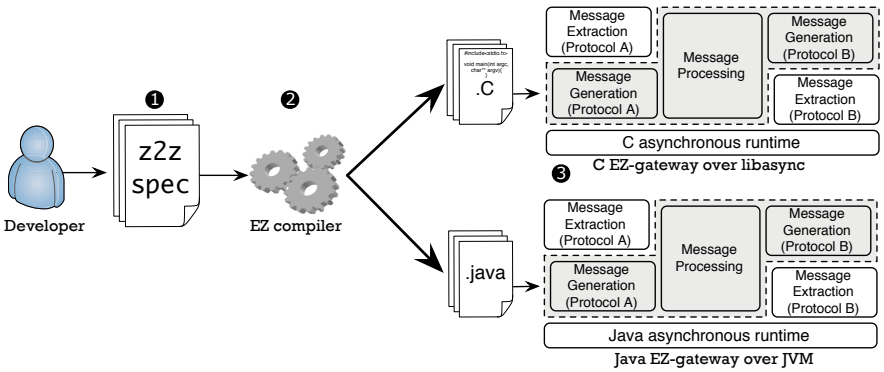

Fig. 1. EZ approach to generate asynchronous multi-platform gateway

2.1 Z2z Domain Specific Language

Our EZ approach has been design especially to be fully compliant with the z2z language that has been proved to be adequate to describe gateways in high level manner by hiding to developers low-level network and system codes.

Z2z Language. The z2z language provides facilities for defining three types of modules: the *Protocol Specification* (PS) modules used to describe the network protocol behaviors, the *Message Specification* (MS) modules used to describe message structures, and the *Message Translation* (MT) modules used to describe the message translation logic. More precisely, a PS module provides information about various properties of the interaction with the network, such as the

transport protocol used, whether requests are sent in unicast or multicast, and whether responses are received synchronously or asynchronously. It also specifies how to dispatch a received request to a specific handler for processing. A MS module defines the useful information to be extracted from incoming messages, i.e., *message views*. It also defines the structure of new messages to be created, i.e., *message templates*. A template contains a message view that describes "holes" to be filled by the translation logic when creating a new message. Finally, a MT module is specified using a dedicated C-like syntax and consists of a set of handlers, one for each kind of relevant incoming requests, as indicated by the protocol specification. It provides domain-specific operators for manipulating and constructing messages, for sending requests and returning responses, and for managing session state across requests.

Z2z Front-End Compiler. Our new `EZ` compiler reuses the `z2z` front-end that deals with the scanning and the parsing of the language and performs consistency checks across the PS, MS and MT modules. For instance, the front-end checks that the MT 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 according to the PS module. It guaranties that all fields and values have been appropriately initialized and used across the different modules. Further, the front-end compiler performs as well a data-flow analysis of the message translation code to detect erroneous specifications and ensure the generation of safe code.

2.2 Asynchronous EZ Runtime

One key contribution of the `EZ` approach is to be able to generate in a transparent manner either C or Java gateways. Specifically, both C and Java based gateways use runtimes implemented with an event-based programming paradigm built on top of the libasync [7] event-driven library for the C version and the new NIO.2 API provided by the JDK7 for the Java version.

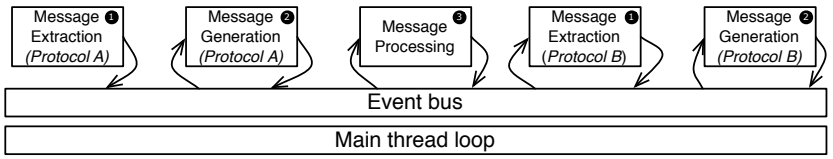

Fig. 2. Event-based processing chain

Gateways are decomposed into key functional building blocks, such as message extraction, processing, and generation. Each of these building blocks registers their interests into network `I/O` events (See Figure 2,❶,❷,❸). Further, each building block may additionally interact with each other *via* the use of events that

they can generate by themselves. For instance, events for either notifying the arrival of a recognized and fully parsed message, for indicating that no data can be read anymore, or for notifying network `I/O` error, are asynchronously dispatch to the building blocks that have declared their interest in these events. Incoming messages that traverse the event-based processing chain are then decomposed into multiple events that are serially dispatched one at a time by a main thread loop. As a consequence, the pipeline, contrary to our previous work [1–5], is not anymore shared among a pool of threads avoiding so mutex lock contentions and increasing inherently performances.

Event-Based Message Processing. As opposed to `z2z` for instance, `EZ` treats each operation performed by the translation logic, that leads to an `I/O` access, as an asynchronous call and thus is compiled by the `EZ` compiler so as to produce a continuation. When the operation is completed, an event is generated and caught by the runtime system that resumes the associated continuation to resume the processing. This strategy results in the creation of many continuations for a handler.

Event-Based Message Extraction. In contrast to the thread model that accumulate data to reconstruct the corresponding message before processing it, `EZ` processes messages even if there are not yet fully parsed. When no more data can be read, because of message fragmentation for example, the parser is paused, the current state is saved and a callback is attached to the availability of new data. When the associated event occurs, the callback is invoked and execution continues in the previously saved state. A soon as a fragment or a message field is recognized by the parser an event is immediately triggered. This event is then stored in the message view to be processed by the other building blocks such as the message processing one. With `EZ`, message parsing drastically limits memory consumption because a message does not need to be entirely saved in memory before starting its parsing. Instead, in C based version of the runtime, two contiguous memory pages are used as a circular buffer, accounting for about 8KB of memory, whereas in the Java version of the runtime two buffers are used and flipped when the first one is exhausted. This only works thanks to the scattered read facility of Java network API. In the either C or Java runtimes, dynamic memory allocation is only required for filling values of the message view when fields are recognized.

Event Based Error Management. When an event occurs, the associated callback function is executed by the runtime system. However, events corresponding to `I/O` operations may never occur if the underlying operation fails to complete, leading thus to memory leaks difficult to address. To overcome this issue, C and Java `EZ` runtimes attach timeout on each event. If the corresponding timer expires before the occurrence of the event, the runtime frees associated resources to prevent memory leaks.

3 Evaluation

To assess the scalability of `z2z` and `EZ` gateways, we have implemented the SMTP/HTTP and HTTP/STMP gateways described previously. Our performance experiments are carried out on a Dell Intel® Xeon® server powered

by 4 processors of 8 hyper-threaded cores clocked at 2.2 GHz. We use the multi-threaded SMTP test client and server distributed with Postfix to stress the generated gateways. C code is compiled using gcc 4.7.2 and Java code executes on HotSpot server 1.7. For our experiments, a given number of simultaneous clients (up to 30) send 1000 messages of 10KB in a closed loop[1] to a SMTP server through a tunneling application. Figure 3 shows the response time for each gateway, as well as the native protocol communication costs (`direct`). Since gateways are I/O intensive, a certain amount of clients are required to get satisfying throughput. The event-based gateways (`EZ-C` and `EZ-Java`) clearly outperform the thread-based gateways (`z2z`).

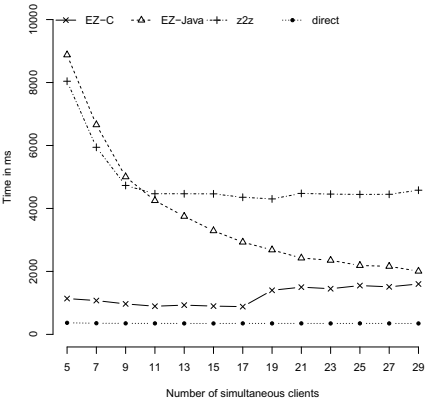

Fig. 3. Time to send 1000 message of 10kB by N clients

4 Related Work

There have been a bulk set of different approaches to protocol interoperability, for instance to name a few, ReMMoC [8], RUNES [9], MUSDAC [10], BASE [11], INDISS [1], Starlink [3] and Enterprise Service Buses [12]. Compared to `EZ`, these approaches have three major weak points: neither they address the difficulty of gateway development nor they tackle the scalability issue, and nor they target both C and Java environment. The closest approach to ours is z2z [2], which constitutes the first generative approach for building gateways. However, z2z generated gateways exhibit poor scalability in the face of the increasing load generated by clients and the increasing size of messages. Furthermore, z2z only targets C environment and is therefore less pervasive than a Java based version.

Scalability has been a major concern in the field of Web servers as these systems must efficiently operate on thousands of files and connections. However, Web servers are CPU intensive oriented while gateways are I/O intensive. Thus experiences on Web server architecture should not be granted for gateways.

5 Conclusion and Future Work

Network protocol gateways are a key enabler of the future of IoT but present a cumbersome challenge for developers. In particular there are three main challenges that need to be overcome to build gateways: (i) providing an easy way to build gateways by hiding to developers intricacies of low-level network and

[1] A client sends a new message only when it receives an acknowledgement from the server for the reception of the previous message.

system code, (ii) tackling the scalability issue,(iii) beeing pervasive, i.e. multi-platform compliant (i.e. C or Java based). As a future direction, multicore architectures are today a reality in all kinds of computing systems, ranging from powerful servers to desktop environments and embedded systems. However current systems and applications are unable to fully exploit these new architectures. Taking advantage of multicore hardware is thus today one of the most important scientific challenges in the systems domain. We are investigating the extension of `EZ` to support multicore architectures. Another potential research direction is to raise on our ongoing research work on hardware accelerated parsers to speed up drastically messages processing in embedded platforms [13].

References

1. Bromberg, Y.-D., Issarny, V.: INDISS: Interoperable discovery system for networked services. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 164–183. Springer, Heidelberg (2005)
2. Bromberg, Y.-D., Réveillère, L., Lawall, J.L., Muller, G.: Automatic generation of network protocol gateways. In: Bacon, J.M., Cooper, B.F. (eds.) Middleware 2009. LNCS, vol. 5896, pp. 21–41. Springer, Heidelberg (2009)
3. Bromberg, Y.D., Grace, P., Reveillere, L.: Starlink: Runtime interoperability between heterogeneous middleware protocols. In: ICDCS, pp. 446–455 (2011)
4. Issarny, V., Bennaceur, A., Bromberg, Y.D.: Middleware-layer connector synthesis: Beyond state of the art in middleware interoperability. In: SFM, pp. 217–255 (2011)
5. Rodrigues, P., Bromberg, Y.-D., Réveillère, L., Négru, D.: ZigZag: A middleware for service discovery in future internet. In: Göschka, K.M., Haridi, S. (eds.) DAIS 2012. LNCS, vol. 7272, pp. 208–221. Springer, Heidelberg (2012)
6. Burgy, L., Reveillere, L., Lawall, J., Muller, G.: Zebu: A language-based approach for network protocol message processing. IEEE Transactions on Software Engineering 37(4), 575–591 (2011)
7. Mazières, D.: A toolkit for user-level file systems. In: USENIX-ATC, pp. 261–274 (2001)
8. 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), 2–14 (2005)
9. Costa, P., Coulson, G., Mascolo, C., Mottola, L., Picco, G.P., Zachariadis, S.: Reconfigurable component-based middleware for networked embedded systems. International Journal of Wireless Information Networks 14(2), 149–162 (2007)
10. Raverdy, P.G., Issarny, V., Chibout, R., de La Chapelle, A.: A multi-protocol approach to service discovery and access in pervasive environments. In: MobiQuitus, pp. 1–9 (2006)
11. Becker, C., Schiele, G., Gubbels, H., Rothermel, K.: Base: A micro-broker-based middleware for pervasive computing. In: PERCOM, p. 443 (2003)
12. Chappell, D.: Enterprise Service Bus. O'Reilly (2004)
13. Mercadal, J., Réveillère, L., Bromberg, Y.D., Gal, B.L., Bissyandé, T.F., Solanki, J.: Zebra: Building efficient network message parsers for embedded systems. Embedded Systems Letters 4(3), 69–72 (2012)