Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Master Seminar I

Raphael Hiesgen

Loosely Coupled Communication in Actor Systems

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Raphael Hiesgen

Loosely Coupled Communication in Actor Systems

im Studiengang Master Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Abgegeben am March 10, 2014

Contents

1 Introduction **1**

2 Introducing Actors **2**
2.1 Erlang 2
2.2 Akka 2
2.3 `libcppa` 3

3 Application Areas for Loosely Coupled Communication **3**
3.1 Social-Area Framework for Early Security Triggers at Airports 3
3.2 The Internet of Things 4
3.2.1 IPv6 over Low-power Wireless Personal Area Networks 4
3.2.2 Datagram Transport Layer Security 5
3.2.3 The Constrained Application Protocol 5
3.3 Internet-wide Systems 6

4 Loosely Coupled Communication **7**
4.1 Communication 7
4.2 Fault-tolerance 8
4.3 Security 8

5 Related Work **9**
5.1 Key Players 9
5.2 Conferences & Groups 9

6 Conclusion **10**

1 Introduction

The classic approach to program concurrent systems uses threads to parallelize the execution of a single program and locks to avoid race conditions. Since the developer is responsible for avoiding race conditions and deadlocks, this programming model is inherently error prone. A higher level of abstraction is offered by the actor model. It describes isolated entities called actors that communicate via message passing. Communication between actors is not limited to local nodes and can be used transparently in distributed systems.

An important characteristic of actor systems is the failure semantic. It defines a basic model of monitors and links, which allow the creation of hierarchical, fault-tolerant systems. Whenever a monitored actor terminates for a non-normal reason, the runtime sends a message containing a failure reason to actors that monitored it. This allows actors to react to failures, e.g., by redistributing work or creating new actors. A link is a bidirectional monitor, enabling developers to define systems in which either all actors are alive or fail collectively.

Today, the Internet connects heterogenous networks that scale from locally distributed low-powered nodes to globally distributed high-powered nodes. Participants in these system do not have the strong coupling that was originally addressed by the actor model. The goal of distributed systems is to solve problems in parallel by splitting them in smaller subproblems. The communication inherent to actors does not scale well using the end-to-end paradigm of the Internet. Simply creating a mesh between all participants is not an option due to the inherent scaling problems of this approach. Instead, a new communication model is required to allow scaleable communication for actors through the Internet. Since nodes are often placed behind NATs and firewalls, a mechanism to pass though them transparently is required.

For connections in constrained environments as well as for connections on the Internet, connection failures are an issue. Although the reasons may be different, from infrastructure failure over unreliable connections to topology changes, we want to recover from such failures and offer reliable communication even in the presence of temporary disconnects.

In order to scale from local up to globally distributed systems, we need to rethink the way actors communicate. The challenges in defining a new communication model are: (1) handle unreliable connections & infrastructure failure, (2) bypass NATs and firewall transparently, (3) provide error propagation in non-hierarchical system and (4) provide secure & authenticated communication.

2 Introducing Actors

In a historic perspective, extending message passing by adding error handling capabilities for distributed systems lead to the actor model. It was first specified by Hewitt et al. in 1973 [1]. 13 years later Agha continued to work out the theoretical aspects of the actor model in his dissertation [2]. At the same time, Armstrong took a more practical approach by developing Erlang [3]. Although Erlang does not mention the actor model, its processes are a de-facto actor implementation.

Actors are concurrent, isolated entities that interact via message passing [2]. They can address each other using network transparent, unique identifiers. In response to a received message, an actor can send messages to other actors, spawn new actors, or change its behavior by exchanging the message handler for processing the next incoming message.

The actor model offers several benefits. Since actors can only interact via message passing, they neither share nor can corrupt each others state. Hence, race conditions are avoided by design. Actor creation is a lightweight operation and is used to distribute work. To handle errors in distributed systems, actors can monitor each other. If an actors dies unexpectedly, the runtime environment sends a message to each actor monitoring it.

2.1 Erlang

Erlang is a concurrent, dynamically typed programming language developed for programming large-scale, fault-tolerant systems [4]. Though Erlang was not build with the actor model in mind, it satisfies its characteristics. Each process in Erlang is in fact an actor with the characteristics described in Section **??**. New processes are created by a function called `spawn`. Their communication is based on asynchronous message passing. Processes use pattern matching to identify incoming messages.

2.2 Akka

Akka[1] is a middleware framework for Scala and Java that implements the actor model. Inspired by Erlang, Jonas Bonér released Akka in 2009 to offer an alternative approach to programming concurrent systems. Similar to Erlang, it aims at the development of reliable distributed applications. Akka recently replaced the original actor implementation in Scala, which was created by Philipp Haller in 2006. As of version 2.10 Akka is part of the Scala standard library.

[1] `http://akka.io`

2.3 libcppa

libcppa[2] is an actor library written in C++11 [5]. It addresses concurrency and distribution by providing a message-oriented programming model. The API is designed in a style familiar to C++ developers and provides a domain-specific language (DSL) for actor programming.

Actors are created using the function spawn. It takes a function or class as first argument and returns a handle to the created actor. The handle can be used to address it and provides a unique address in a distributed system. Actors can communicate via asynchronous or synchronous message passing, using the send or sync_send functions. Network transparency hides whether an actors runs on the same core, same system, or another machine in the network.

The behavior of an actor specifies its response to messages it receives. libcppa uses partial functions as message handlers, which are implemented using pattern matching. Messages that cannot be matched stay in the buffer until they are discarded manually or handled by another behavior. The behavior can be changed dynamically during message processing.

3 Application Areas for Loosely Coupled Communication

The first part describes the Internet of Things (IoT), a network that consists of many low-powered nodes, and the second part describes Internet-wide systems, a global scale network that includes work-stations as well as servers and mobile nodes.

3.1 Social-Area Framework for Early Security Triggers at Airports

The project Social-Area Framework for Early Security Triggers at Airports (SAFEST[3]) [6] is a Franco-German cooperation. The project aims to create an area surveillance system deployed at the airport Berlin Brandenburg. Distributed sensor nodes collect anonymized data and forward it towards a central evaluation. Images are partially processed by nodes to prevent redundant data and use the available bandwidth efficiently. If the system detects a panic situation among the masses it alerts the operator. Key aspects are the respect of privacy and resilience against malfunction. To enabled a high-level modeling & development, the software will be developed using libcppa and deployed on nodes running RIOT[4] [7].

[2] http://libcppa.org
[3] http://safest.realmv6.org/
[4] http://www.riot-os.org

3.2 The Internet of Things

The Internet of Things (IoT) describes a network of low-powered nodes, which are connected to perform a common task. While individual nodes can only perform simple tasks, the network enables the processing of complex tasks. As a result, a highly distributed workflow is common in this area.

Applications in the IoT include sensor networks in public or disaster areas that help acquire an overview of the situation, as well as smart home systems. Furthermore, sensor systems enable machines to upload data to the Internet, a task that was reserved to human interaction before. Enable us to keep track of health and home through smartphones and webpages.

Having access to a wide range of data, security concerns are inevitable when building such systems. Wireless communication can be received by everyone in the vicinity and as a result should be encrypted. In addition, the insertion false message is of equal threat and can be addressed by the authentication of trusted nodes. Authentication also includes a model to revoke trust from nodes, if an intrusion is detected.

Failures are not only a result of malicious actions. Interferences are caused by communication between nodes in the same system or in the vicinity. Furthermore, nodes mail fail due to limited battery life and other hardware errors. A system build under these conditions needs mechanics to retain functionality and shift tasks in case components of the system fail.

Traditional programming models do not address this environment, as they favor many core machines with a strong coupling between nodes. The fallback makes use of low-level programming languages such as `C`, which require manual handling of network communication and synchronization. This introduces a lot of complexity and thus leads to more error sources. `libcppa` allows native development in a `C++` environment to address this problem and develop software from a high abstraction layer. While the communication in `libcppa` is currently build with the strong coupling known from traditional actor systems, the work aims to add capabilities to adapt to the requirements of loosely coupled communication.

3.2.1 IPv6 over Low-power Wireless Personal Area Networks

IPv6 over Low-power Wireless Personal Area Networks (6LoWPAN) [8] enables Internet connectivity on constrained nodes by simplifying IPv6 functionality and optimizing related protocols. It is often used in conjunction with IEEE 802.15.4 [9], which specifies wireless embedded radio communication. The traditional Internet protocols require an effort that can not be met by most embedded devices, such as TCP, SNMP and a minimum frame size of

1280 bytes [10]. 6LoWPAN addresses these problems while maintaining compatibility to IPv6. A translation between 6LoWPAN and IPv6 can be done stateless by edge routers.

Instead of the minimum 1280 bytes size required by IPv6, IEEE 802.15.4 packets have a size of 127 bytes. With 25 bytes frame overhead, 102 bytes are available for IP, transport protocol and payload. With 40 bytes IP header and 8 bytes UDP header the payload can have a maximum of 53 bytes. 6LoWPAN includes a header compression to allow a payload of up to 108 bytes.

3.2.2 Datagram Transport Layer Security

The Datagram Transport Layer Security protocol (DTLS) [11] provides features of the Transport Layer Security protocol (TLS) for datagram protocols. Hence, it has to handle message loss or packet reordering. Similar to TLS, DTLS provides encryption and integrity.

The DTLS header as been extended with a sequence number to allow packet reordering. Furthermore DTLS implements a retransmission timer to handle packet loss during the initial handshake. Although it does not allow fragmentation in normal messages, the handshake requires fragmentation as it does not fit into a single datagram. To allow this, the header also includes a offset and length.

3.2.3 The Constrained Application Protocol

A protocol designed for the use in the IoT and machine-to-machine (M2M) communication is the Constrained Application Protocol (CoAP) [12]. It defines a request-response model adapted from HTTP, but tries to avoid its complexity. As such, it implements the GET, PUT, POST and DELETE methods known from HTTP. However, CoAP is designed to work asynchronously and to be used via datagram protocols, such as UDP. It can be used with DTLS to provide secure communication.

Request can contain a method code to request an action or a URI to identify a resource, while responses contain a response code and possibly a resource. Four messages types are defined: Confirmable (CON), Non-confirmable (NON), Acknowledgement (ACK), Reset (RST). Requests can be contained in CON and NON messages. Responses can be contained in the same messages types as well as piggy-backed in ACK. CoAP integrates two layers in a single protocol.

The messaging layer deals with datagrams and their asynchronous nature. In both, requests and responses, a 4 bytes binary header is followed by compact binary options and a payload. Messages contain a 2 byte message ID for duplicate detection. The ID is also used to provide reliability in CON messages, which are retransmitted after a timeout until an ACK with the

same message ID is received. RST messages are used to respond to messages that could not be processed.

The request-response layer is embedded in the messaging layer. Each message contains either a method or response code, optionally request or response information and a token that is used to match responses to requests independently of the messaging layer. A single request may lead to multiple responses, e.g., when using multicast.

CoAP defines a mechanism to discovers servers that offer CoAP services. Either a node learns a URI of a resource handled by a server or an "all-CoAP-nodes" multicast group can be used for the discovery.

3.3 Internet-wide Systems

This groups contains nodes of various size and performance with an internet connection as a common component. Hence, these systems may be distributed on a global scale. They face reliability problems due to temporary infrastructure failure or movement, should transparently handle obstacles in form of NATs and firewalls, and avoid ongoing connections.

The Hypertext Transfer Protocol (HTTP) [13] is a widely used request-response protocol that has desirable characteristics for loosely coupled communication. It is stateless, meaning that each request-response pair is an independent transaction. Furthermore, some of the methods in HTTP are defines as safe or idempotent. Safe methods do not change servers state. Idempotent methods can be multiple identical requests lead to the same result as a single request. HTTP is usually used over TCP/IP on port 80.

An example for a widely used protocol to create a session between multiple parties is the Session Initiation Protocol (SIP) [14]. It allows the creation, termination and modification of sessions and is often used to for multimedia applications. SIP addresses participants with a Uniform Resource Identifier (URI) [15] and enables their discovery via proxy servers.
Session are created by a transaction based request/response model. When a session is initiated, messages are forwarded by the proxies between the participants. Once the setup is finished, a direct connection is established. SIP does not include information about the session, rather another protocol format must be used to describe it. The Session Description Protocol (SDP) [16] is such a format and can be carried in the body of SIP messages.

Establishing a connection is further hindered by NATs and firewalls. It is possible to detect ones public IP address using Session Traversal Utilities for NAT (STUN) [17]. The protocol defines a STUN request that is sent to a STUN server and answered with a packet containing IP address and port as seen by the server. STUN servers can be found with a DNS query. In addition, STUN provides a way to keep NAT bindings alive, which are otherwise dropped after a timeout. An extension of SIP uses STUN to enable NAT traversal (SIP Outbound)

[18], which basically keeps the connection to the registrar alive and uses it to enable a bidirectional flow.

It is possible to "punch holes" [19] in some NAT implementations. This enable a direct communication between multiple nodes in NAT networks. In cases where this is not possible, Traversal Using Relays around NAT (TURN) [20] defines an extension to STUN, which allows the use of a relay server to enable communication between such clients.

Another protocol that makes use of STUN is Interactive Connectivity Establishment (ICE) [21]. ICE is build of top of an offer-answer exchange model, such as provided by SDP. Nodes that want to connect using ICE are required to communicate via another protocol, e.g., SIP with allows the offer-answer exchange in form of SDP. Both nodes collect their available address/port combinations, i.e. from their own interface, the NAT and/or a TURN server. These combinations are exchanged via the offer-answer protocol. ICE proceeds to pair the address and use each pair for a STUN request. An address/port pair that works in both ways, can be use for communication.

4 Loosely Coupled Communication

The systems we examine are inherently distributed. However, their scales range from local settings to global distribution. In any case, these system should work in a decentralized way and thus prevent strong ties to a single instance. Another requirement is scalability. Ideally an increase in cores or nodes leads to a better performance, such as more processing power or a better coverage by sensors. We examine communication in these systems with regard to the following aspects: communication, fault tolerance and security.

4.1 Communication

We focus on two communication use cases in this work: long-distance traffic in Internet-wide systems and low-powered communication in the Internet of Things, as presented in Section 3. Although both rely on IP, the IoT usually uses 6LoWPAN (over 802.15.4) which add constraints for packet size and processing power.

On the transport layer we usually use TCP or UDP. TCP is a stateful protocol, that offers reliable transport of packets as well as maintained packet order. It does have a header size of at least 20 bytes and is not optimized for the use in 6LoWPAN environments, e.g., TCP can not differentiate between packet loss due to congestion and link failure. UDP on the other hand, offers no reliability for packet delivery or order, which have to be provided by the application layer if needed. However, it is stateless and has a small header of 8 bytes.

We have examined application protocols in earlier sections. HTTP is belongs in this cat-

egory. It is based on an request-response model and expects a reliable transport protocol, such as TCP. However, from an applications perspective HTTP is stateless, meaning that each request is an independent transaction. CoAP, which was presented in Section 3.2.3, is an adaption of HTTP that is optimized for constrained environments and can be used on with UDP. It offers optional reliable message transport.

We will take a look at an example message exchange in `libcppa`. To keep the message size small, `libcppa` does not send type information with each message, but uses tokens to tag data with the types. For this to work, each actors must have the same mapping. When communication is initiated, participating actors exchange their type information to ensure this. Hence, it is import for future interactions that this exchange works reliably. The following list present three approaches to handle this exchange:

Setup Phase Create a TCP connection, or a similar protocol, to exchange the information and switch to a lightweight protocol for subsequent messages. During the implementation of DTLS a similar consideration had to be made concerning the initial handshake [22]. The developers decided against the use of a second protocol.

Lazy Instead of exchanging type information ahead, an actor requests information once it encounters types it can not handle.

Remove Token Instead of a token, the type information are included in each message. This leads to overhead, but allows stateless communication.

4.2 Fault-tolerance

Fault tolerance describes a systems resilience to hardware or software failure within. Of interest is the systems ability to continue operation if some components fail, e.g., by recovering failed tasks and redistributing the work among the available components. An ideal solution would not require a central instance and only affect nodes in the vicinity.

4.3 Security

The architectures we address are wide open. Hence, messages can originate from, received by and potentially be altered by any source in the network. On the Internet this could be a connected node and in a wireless network this could be a node in the vicinity. We want to provided confidentiality, integrity and authenticity for the message exchange between our nodes. A key aspect is the avoidance of a central trusted instance, since it creates a strong coupling. At the moment, `libcppa` does not provide encrypted communication or authentication.

5 Related Work

This section examines the persons and conferences in this research area. The first section takes a look at the key players and the second section takes a look at upcoming conferences.

5.1 Key Players

Carl Hewitt Initial paper on actors, worked on actor model in 1973

Gul Agha Doctoral student of Hewitt, employed at the University of Illinois, on the steering committee of the Agere

Joe Armstrong Developed Erlang in 1986

Jonas Bonér, Viktor Klang, Martin Odersky Work at Typesafe[5], a company that is focused developing software to build reactive applications. They defines reactive through applications to be event-driver, scalable, resilient and responsive. Besides Scala and Akka, they develop the Play framework.

Carsten Bormann, Zach Shelby Development of 6LoWPAN & CoAP in the IETF.

5.2 Conferences & Groups

INFOCOM IEEE International Conference on Computer Communications (April/May)

SIGCOMM Conference ACM Conference of the Special Interest Group on Data Communication (August)

CCS ACM Conference on Computer and Communications Security (November)

WiSec ACM Conference on Security and Privacy in Wireless and Mobile Networks (July)

C++Now Hosted by Boost and ACM, focused on C++ language development as well as application and design (May)

Splash ACM conference on Systems, Programming, Languages and Applications. Featured the Agere Workshop on Programming based on Actors, Agents and Decentralized Control in the past (October)

ICDCS International Conference on Distributed Computing Systems (June/July)

[5] `http://typesafe.com`

PODC ACM Symposium on Principles of Distributed Computing (July)

USENIX OSDI USENIX Symposium on Operating Systems Design and Implementation (October)

SIGBED, SIGSAC, SIGPLAN ACM Special Interest Groups on Embedded Systems / Security, Audit and Control / Programming Languages

TOSN, TON ACM / IEEE journals on Transactions on Sensor Networks / Networking

6 Conclusion

In this work, we examined the requirements to adapt the communication in actor systems to the challenges introduced by Internet of Things and Internet-wide systems.

Although both areas are build on top of IP, the IoT introduces additional constrains through the restricted platforms in use. Participants are low-powered and usually communicate over lossy networks. Protocols designed for these constrained environments take these characteristics into account. CoAP, which is adapted from HTTP, does not rely on a reliable transport layer and includes mechanisms to provide reliable message transport when needed. Security can achieved by DTLS, an adaption of TLS for datagrams. It is also designed to work without the reliability of the transport layer. CoAP also has specification for usage with DTLS. In combination, these protocols protocols provide encrypted, stateless and optionally reliable communication between nodes in the IoT.

Some issues are specific to the Internet and do not show up in the IoT. NATs and firewalls are not found in wireless sensor networks, but are widespread components in the Internet. We examined STUN and ICE, which aim to allow communication through these obstacles. While STUN provides simple tools to detect ones public address and keep NAT bindings alive, ICE systematically finds ways to bypass NATs. However, it requires an exchange via an offer-answer protocol such as SDP.

Furthermore, we have identified a need to remodel the propagation of errors in non-hierarchical systems as well as new concepts for decentralized authentication. These challenged need further research and are thus left to future work.

In summary, the examined protocols allow us to approach some of our challenges, such as handling unreliable connections and bypassing NATs. However, we still need to combine these findings with the actor model and determine how suited these solutions are in practice.

References

[1] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proceedings of the 3rd IJCAI*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245.

[2] G. Agha, "Actors: A Model Of Concurrent Computation In Distributed Systems," MIT, Cambridge, MA, USA, Tech. Rep. 844, 1986.

[3] J. Armstrong, "A History of Erlang," in *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*. New York, NY, USA: ACM, 2007, pp. 6–1–6–26.

[4] ——, "Making Reliable Distributed Systems in the Presence of Software Errors," Ph.D. dissertation, Department of Microelectronics and Information Technology, KTH, Sweden, 2003.

[5] D. Charousset and T. C. Schmidt, "libcppa - Designing an Actor Semantic for C++11," in *Proc. of C++Now*, May 2013.

[6] E. Baccelli, G. Bartl, A. Danilkina, V. Ebner, F. Gendry, C. Guettier, O. Hahm, U. Kriegel, G. Hege, M. Palkow, H. Pertersen, T. Schmidt, A. Voisard, M. Wählisch, and H. Ziegler, "Area & Perimeter Surveillance in SAFEST using Sensors and the Internet of Things," in *Workshop Interdisciplinaire sur la Sécurité Globale (WISG2014)*, Troyes, France, Jan. 2014. [Online]. Available: http://hal.inria.fr/hal-00944907

[7] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "RIOT OS: Towards an OS for the Internet of Things," in *Proc. of the 32nd IEEE INFOCOM. Poster*. Piscataway, NJ, USA: IEEE Press, 2013.

[8] N. Kushalnagar, G. Montenegro, and C. Schumacher, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals," IETF, RFC 4919, August 2007.

[9] J. Hui and P. Thubert, "Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks," IETF, RFC 6282, September 2011.

[10] Z. Shelby and C. Bormann, *6LoWPAN: The Wireless Embedded Internet*, 1st ed. Wiley Publishing, 2009.

[11] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2," IETF, RFC 6347, January 2012.

[12] Z. Shelby, K. Hartke, and C. Bormann, "Constrained Application Protocol (CoAP)," IETF, Internet-Draft – work in progress 18, June 2013.

[13] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1," IETF, RFC 2616, June 1999.

[14] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," IETF, RFC 3261, June 2002.

[15] T. Berners-Lee, R. T. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," IETF, RFC 3986, January 2005.

[16] M. Handley and V. Jacobson, "SDP: Session Description Protocol," IETF, RFC 2327, April 1998.

[17] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session Traversal Utilities for NAT (STUN)," IETF, RFC 5389, October 2008.

[18] C. Jennings, R. Mahy, and F. Audet, "Managing Client-Initiated Connections in the Session Initiation Protocol (SIP)," IETF, RFC 5626, October 2009.

[19] P. Srisuresh, B. Ford, and D. Kegel, "State of Peer-to-Peer (P2P) Communication across Network Address Translators (NATs)," IETF, RFC 5128, March 2008.

[20] R. Mahy, P. Matthews, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)," IETF, RFC 5766, April 2010.

[21] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," IETF, RFC 5245, April 2010.

[22] N. Modadugu and E. Rescorla, "The Design and Implementation of Datagram TLS," in *In Proc. NDSS*, 2004.