A Pattern Language for Application-level Communication Protocols

Jorge Edison Lascano[1,2], Stephen Wright Clyde[1]
[1]Computer Science Department, Utah State University, Logan, Utah, USA
[2]Departamento de Ciencias de la Computación, Universidad de las Fuerzas Armadas ESPE, Sangolquí, Ecuador
email: edison_lascano@yahoo.com, Stephen.Clyde@usu.edu

***Abstract*—Distributed applications depend on application-layer communication protocols to exchange data among processes and coordinate distributed operations, independent of underlying communication subsystems and lower level protocols. Since such protocols are application-specific, developers often must invent or re-invent solutions to reoccurring problems involving sending and receiving messages to meet specific functionality, efficiency, distribution, reliability, and security requirements. This paper introduces a pattern language, called CommDP, consisting of nine design patterns that can help developers understand existing reusable solutions and how those solutions might apply to their situations. Consistent with other pattern languages, the CommDP patterns are described in terms of the problems they address, their contexts, and solutions. The problems and consequences of the solutions are evaluated against four desirable qualities: reliability, synchronicity, longevity, and adaptability for scalable distribution.**

Keywords-design patterns; pattern languages; communication protocols.

I. INTRODUCTION

At the application level, a distributed system is two or more processes sharing resources and working together via network communications to accomplish a common goal [1][2]. Such systems are ubiquitous in today's Internet-connected world and are found in virtually every application domain, such as personal productivity tools, social media, entertainment, research, and business. Even single-user software systems that appear to be non-distributed may in fact communicate with other processes in the background to download updates, track usage statistics, or capture error logs, and are therefore actually distributed systems.

In general, the developers of a distributed system try to increase its overall throughput, reliability, and scalability by hosting data and/or operations on multiple machines, while minimizing network traffic, congestion, and turn-around times. Exactly how they do this depends heavily on the nature and requirements of the application. In some cases, developers may choose to distribute instances of one type of resource, e.g., image files in a peer-to-peer shared photo library. In other situations, developers may group resources such that all instances of a single type are on one server. Still in other cases, developers can take hybrid approaches, distributing certain types of resources among peers and hosting other types on dedicated servers. A closely related design issue deals with the granularity of the distributed resources, i.e., data and operations. From a data perspective, the possible choices range from whole databases to individual records or even individual fields within records. From an operations perspective, the choices range from entire subsystems to atomic operations. With today's programming languages, many developers follow the object-oriented paradigm, encapsulating operations with data and making choices for granularity that range from entire sets of objects to object fragments [3].

Besides deciding on the granularity and distribution of resources (data, operations, or objects), developers often have to consider requirements for security, fault tolerance, maintainability, openness, extensibility, scalability, and dynamic quality of service [2]. The degree to which an application possesses these desirable characteristics is primarily a consequence of architectural design choices, which, in turn, place new requirements on inter-process communications.

The problem is not that existing application-level communications protocols are poorly designed and implemented; rather, the problem is that application developer has to re-invent or re-design them for every new application.

In this paper, we will refer to an exchange among two or more processes for a particular purpose as a *conversation*. A single conversation may be short and simple, like querying a stock's price, or it could be long and complex, like the streaming of a video. The rules that govern a particular type of conversation are a *communication protocol* and a collection of protocols is called a *protocol suite* [4][5].

Application-layer communication protocols (ACPs) are often defined on top of other protocols. For example, the *Hypertext Transfer Protocol* (HTTP), which is an ACP, is defined on top of the *Transmission Control Protocol* (TCP) [1]. Many higher level ACPs, like webservice-based ACPs, are in turn defined on top of HTTP [1]. Section II provides additional background on protocols and protocol suites, as well as a brief discussion on layered communication subsystems.

Because requirements for ACPs can come from an application's (a) functional requirements, (b) architectural design, and (c) use of lower layer protocols, coming up with effective designs can be challenging. Fortunately, the problems that developers are likely to encounter are not uncommon and have known solutions. The key is to capture this knowledge in a way that developers can easily find it and adapt it to a new application. This is precisely what design patterns can do [6].

Unfortunately, design patterns for communication protocols at application layer have yet not been gathered, correlated, and formally organized into a cohesive and thorough collection. To this end, this paper introduces a system of design patterns, i.e., a pattern language, for ACPs,

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-498-5

called CommDP. The patterns in CommDP come from a variety of sources and are by themselves not new ideas, as is the case for all newly documented design patterns [7]. Section III provides more background information on design patterns and pattern languages, as well as information about related work.

Since designing ACPs is different from designing executable software, it is necessary to discuss desirable qualities for protocols. Section IV introduces four, namely reliability, synchronicity, longevity, and adaptability for scalable distribution. Section V presents a design pattern template that incorporates these characteristics into the definition of communication problems and the consequences of pattern solutions.

Section VI-A introduces three communication idioms that act as conceptual building blocks for all the ACP patterns in CommDP. We then provide an overview of the ACP patterns in Section VI-B. Additional details for the CommDP patterns are available on-line[1].

Patterns are rarely used in isolation; instead, developers typically weave multiple pattern instantiations together to create complete solutions [8]. A system of patterns, i.e., a pattern language, not only includes a collection of patterns, but relationships among them that help developers know how they might be effectively combined [9]. Section VI-C provides a digest of these relationships for CommDP. Finally, in Section VII, we summarize the value of CommDP and outline our future research direction.

II. Protocols and Protocol Suites

Software and electrical engineers model, design, and implement inter-process communications in layers. Fig. 1 shows a simple 5-layer model commonly favored by those

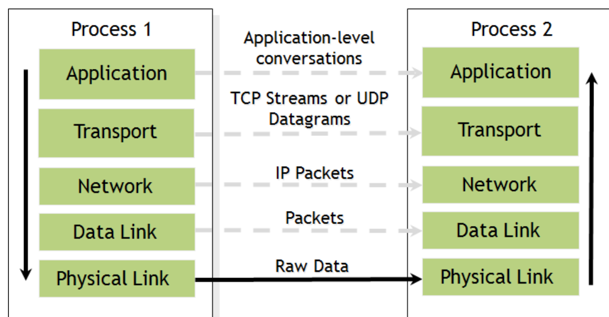

Figure 1. 5-Layer Model for IP-based Communications

who work with IP-based protocols [1][10]. There are several other common models, such as the 7-layer OSI model [11]. A conversation between Process 1 and Process 2 can be discussed at any layer and, for each layer, it must adhere to agree upon protocol(s) for that layer. For example, if Process 1 were a web browser, Process 2 were a web server, and the conversation a simple web-page request, then the application-layer protocol would be HTTP, the Transport-layer protocol would be TCP, and the Network-layer protocol would be IP.

Besides providing a convenient way for discussing protocols, layered models establish a basis for creating substitutable software communication subsystems. Since we are addressing ACPs in this paper, we do not deal directly with design and implementation of these software components. Nevertheless, we assume that appropriate communication subsystems exist at the transport layer for streaming of unstructured data and transmitting datagrams (semi-structured data). Section V relies on this reasonable assumption to define four communication idioms.

At the application layer, a protocol governs why, when, and how processes interact with each other to accomplish a common goal. Specifically, an ACP should define the following:

1. the processes involved in the interaction in terms of the roles they play during a conversation;
2. the possible sequences of messages for valid conversations;
3. the structure of the messages;
4. the meaning of the messages; and
5. relevant behaviors of the participating processes.

Because processes in a distributed system typically have to communicate with each other for many different tasks, e.g., authentication, resource sharing, and coordination, it is common for a distributed system to require multiple ACPs, i.e., an ACP suite.

III. Design Patterns and Pattern Languages

Christopher Alexander et al. defined a pattern as a reusable solution to a reoccurring problem [9]. Kent Beck and Ward Cthe detailsunningham started to apply the concept of pattern languages to software engineering in 1987 [12], and the idea was later popularized by Eric Gamma et al. with their landmark language of 23 patterns [6]. Since then, pattern languages have been documented for many areas of software engineering, including architectural design [13][14], user-interface design [15], event handling [16][17][18], and concurrency [19]-[25]. There are even patterns specifically for distributed computing [8], distributed objects [26][27][28], communication software [29][30], RESTful and SOAP web services [31], cloud computing [32], and distributed real-time and embedded systems [33]. However, to date, no pattern language has been published specifically for ACPs.

There are two hoped-for benefits of pattern languages that are important to ACP design. First, they create a vocabulary that enables developers to discuss complex ideas in a few words [28]. Second, they allow developers of all experience levels to benefit from expert reusable solutions [34].

IV. Qualities of Communication protocols

Like software, ACP suites, as whole, should possess certain desirable qualities that contribute to the overall success of a system. Some of these desirable qualities come directly

[1] http://commdp.serv.usu.edu/

from the software arena. For example, cohesion is the degree to which the elements of a software component align with a single purpose [35]. Cohesion and its definition can apply almost directly to ACPs, but this is a subject for future research (see Section VII). Another desirable software quality directly applicable to ACPs is modularization. Modularization is the degree to which a system is divided up into independent components [36]. When a system has good modularization, developers do not have to look very far beyond a component to understand it or reason about it. We believe the same to be true for ACPs, but the details of modularization applied to ACPs are also a subject for another paper (see Section VII).

Although we believe cohesion and modularization are important qualities for ACPs, they do not directly help in describing reoccurring communication problems nor are they good discriminators for reusable solutions, because all pattern solutions should, by definition, have good cohesion and modularization. So, we turn our attention to four other qualities with discriminating definitions for ACPs, namely: reliability, synchronicity, longevity, and adaptability for scalable distribution.

A. Reliability

For an ACP, reliability is the degree to which a process that sends a message as part of a conversation obtains an assurance that the intended recipient(s) received it, entirety and uncorrupted, and reacted as prescribed in the ACP. At the application level, reliability is typically achieved by the recipients returning messages that provide the sender with confirmation that the message was received and/or processed. When such return messages fail to arrive in a timely fashion, reliable ACPs will require the sender to retransmit the original message.

In Section VI, where we present an overview of the ACP patterns in CommDP, we rank each of the patterns in terms of reliability using the following 3-point rubric:

Rank/Criteria

3 The problem (P) addressed by the pattern is primarily concerned with reliability and the solution (S) can make the following guarantees under normal and extreme conditions:
   a. The sender can distinguish between successful and failed conversations.
   b. The receiver can distinguish between successful and failed conversations.
   c. In successful conversations, any process X that sends a message M to process Y, gives a timely assurance to X (in some subsequent message) that Y received M.
   d. In successful conversations, for any process X that sends a message M to process Y, if M is supposed to trigger a non-trivial behavior in Y, then X receives a timely assurance that Y successfully handled M.

2 P is concerned with reliability and S can guarantee at least (a) and (c) from above in normal situations.

1 P is not concerned with reliability and S doesn't limit reliability.

Clearly, there are other conceivable problem/solution criteria for reliability not listed above, such as a reoccurring problem where reliability is a major concern and a solution that doesn't address it. However, we don't include such meaningless classifications because they wouldn't help classify patterns with expert, reusable solutions.

B. Synchronicity

In the most general sense, synchronization deals with the coordinated execution of actions in a distributed system and what the state information is necessary for that coordination. This broad definition encompasses, but is not limited to, the common view among programmers that synchronous communications occur when the sender of a message stops and waits for a response from the message receiver [37]. However, this is not the only way to achieve synchronization. Some other common mechanisms are logical clocks [38][39], vector clocks [40][41], vector timestamps [42], optimistic concurrency controls [43], and timing signals.

To evaluate the synchronization requirements for ACPs, we consider: (a) what are the actions that need to be coordinated, (b) where will those actions be executed, and (c) what kind of state information is needed to achieve the desired coordination. A distributed system may perform many different tasks comprised of numerous operations, but rarely all of them have to be fully coordinated. In fact, the more independent the individual operations are, the more a system can maximize concurrency and increase throughput. From a coordination perspective, where the operations take place is actually more important than what the operations do. For example, if all of the actions occur in just one process, then that process may not need to know anything about the state of the other process. Once developers know what operations have to be coordinated and where they will execute, they can consider what local or global state information the coordination logic will need.

To rank synchronicity for ACP patterns, we will use the following definitions:

- C is a conversation involving a closed set of processes, $C.P = \{p_1, \dots, p_n\}$, and a set of messages, $C.M = \{m_1, \dots, m_n\}$, such that $sender(m_i) \in C.P \wedge receivers(m_i) \subseteq C.P$ for $1 \leq i \leq n$ where $sender(m_i)$ is the process that sent message m_i and $receivers(m_i)$ is the set of processes that received m_i.
- A is a set of operations, $\{a_1, \dots, a_n\}$ that run on $C.P$ and whose execution requires coordination, e.g., ordering, simultaneous execution, etc.
- $h(a)$ is the host process for operation, a, where $a \in A$ and $h(a) \in C.P$
- $s(a)$ is the state information that $h(a)$ needs to coordinate a's execution with the rest of the operations in A.
- $H(A)$ is the set of host processes for all operations in A

Below is an informal 3-point rubric for ranking synchronicity for CommDP patterns using these definitions. We believe that a more rigorous ranking system would have value beyond the categorization of ACP patterns, and its full definition is beyond the scope and purpose of this paper.

Rank/Criteria

3 The problem (P) addressed by the pattern deals with situations where $|H(A)| > 1$ and the solution (S) can guarantee that for all $a \in A$, h(a) receives s(a) via messages, $m_i \in C.M$, in time to do the prescribed coordination.

2 P deals with situations where $|H(A)| = 1$ and S can guarantee that for all $a \in A$, h(a) receives s(a) via messages, $m_i \in C.M$, in time to do the prescribed coordination.

1 P is not concerned with synchronicity, e.g., $|A| = 0$, and S does not limit synchronicity.

C. *Longevity*

Longevity is the degree to which an ACP can support long-running conversations caused by long-running operations. The primary problem for conversation with long-running operations is that there could be huge span of times when processes are uncertain of each other's states. Consider a simple request/reply conversation where some process A sends a request to B, but B takes a long time to execute the requested operation and sends back a reply. While waiting for the reply, process A doesn't know if B received the request, has failed, or is just taking a long time. ACPs that support long-running operations include mechanisms for exchanging state information independent of results.

We rank the longevity for ACP patterns according to the following 3-point rubric:

Rank/Criteria

3 The problem (P) addressed by the pattern is primarily concerned with long-running conversations and the solution (S) can guarantee the following in successful conversations:
   a. Participants made aware of each other's states in periodically.
   b. Each participant in the conversation can detect when other participants are no longer available or accessible.

2 P is concerned with long-running conversations and S provides for (a).

1 P is not concerned with long-running conversations and S doesn't limit longevity.

D. *Adaptability for Scalable Distribution*

ACPs can support scalability by providing location transparency and/or replication transparency [42], and by allowing resources (data, operations, or objects) to be distributed across multiple hosts. To understand location and replication transparency, consider a website with a large number of resources. It can support scalability by placing the various resources on an expandable collection of backend servers and use a front-end server to distribute requests from browsers. If the browser doesn't need to know where a resource is actually located, then the system supports location transparency. Similarly, as traffic increases, the system could replicate resources across multiple backend servers. If the client doesn't have to know that replicas exist, then the system supports replication transparency. Both location and replication transparency simplify scalability.

Another technique for supporting scalability is allowing complex resources to be broken up into smaller resources and distributed across multiple servers. One approach for doing this is to untangle cross-cutting concerns, like security or logging, from complex operations and host these pieces of functionality on proxies [44][42].

Here is a simple rubric for the adaptability for scalable distribution.

Rank/Criteria

3 The problem (P) addressed by the pattern is primarily concerned about scalability or the distribution of action or resources and the solution (S) can provide two or more of following:
   a. Location transparency for shared resources distributed across multiple hosts
   b. Load balancing with shared resources replicated across multiple hosts
   c. Untangling of cross-cutting concerns into separate actions

2 P is concerned with scalability or distribution of resources and S provides at least one (a), (b), or (c).

1 P is not concerned with scalability or distribution and S doesn't limit them.

V. TEMPLATE FOR COMMUNICATION-PROTOCOL PATTERNS

To document the patterns in CommDP, we have developed a template, loosely based on the way Gamma, Helm, Johnson and Vlissides documented their patterns [34], referred to here as the GoF template. The main goal is to keep the documentation as simple as possible, while still capturing the details of the pattern. Following are the elements of CommDP pattern template.

A. *Name*

As with the GoF template, the name uniquely identifies the pattern. Since the name will become part of the vocabulary for the pattern language, it is important that it captures the essence of the pattern, distinguishes it from other patterns, and is as concise as possible.

B. *Intent*

The intent is an abstract for the pattern. It summarizes the problem, the context, and the solutions, particularly in terms of reliability, synchronicity, longevity, and adaptability for scalable distributes.

C. *Description*

The description consists of three subsections that explain the problem, context, and solution. The problem subsection relates closely to the Motivation part in the GoF template, in that it explains the nature of the reoccurring communication-protocol design problem. This subsection should highlight the problem's need for reliable communications, synchronization, long-running conversations, or scalable distribution. The context subsection is like the Applicability in the GoF template, capturing information when the pattern may or may not be applicable and assumptions about distributed systems in which the communications will take place. The solution is

analogous to the Structure in the GoF template. It focuses on the describing protocol design ideas and how they can be adapted.

D. Consequences

As with the GoF template, the consequences are important part of CommDP pattern definitions because developers will use them to determine if the pattern is a good fit for a particular situation. The consequences of CommDP patterns are described in terms of the qualities discussed in Section 4. The rankings provide a general classification, and pros explain the consequences in more detail.

E. Known Uses

Like the GoF template, this part references known instances of the pattern in production systems

F. Aliases and Related Work

The section combines two elements of the GoF template by the similar names.

G. References

This section contains a bibliography for the citations made elsewhere in the pattern definition.

VI. COMMDP

A design pattern is composed of a set of patterns and idioms that are used together to solve a design engineering problem.

A. ACP Idioms

Before launching into a description of CommDP's patterns, it is important to first introduce three fundamental building blocks for all ACPs: point-to-point send, multicast, and broadcast. These are idioms instead of patterns because their usage depends on the lower layer communication protocols and because, by themselves, they do not address the qualities discussed in Section 4.

A *point-to-point* send is the transmission of a single message from one process to another, such as a message sent over a TCP connection or via a UDP datagram. An underlying communication subsystem may provide some reliability relative to the transmission but, at the application-level, a single message does not allow the sender to know if the receiver processed the message or anything about the receiver's state, nor does it help with longevity or adaptability for scalable distribution.

A *multicast* send is the transmission of a single message to a set of receiving processes [45]. It can be implemented at virtually any layer in communication hierarchy, including the physical layer. Mechanisms for identifying the group of receiving processes vary from sender determined to receiver subscriptions. By themselves, multicast are idioms for ACPs. The same is true for *broadcasts*, which also transmit messages to multiple receivers [45].

B. ACP Patterns

Table I. COMMDP PATTERNS lists the nine patterns currently in CommDP, along with their rankings from their consequences relative to the characteristics discussed in Section 4 (R=Reliability, S=Synchronicity, L=Longevity, and A=Adaptability for Scalable Distribution). Their full definitions are available on [http://commdp.serv.usu.edu].

The *Request-Reply* pattern is undoubtedly the most common. It addresses the problem where a process, A, needs to access or use shared resources in another process, B, with a reasonable degree of reliability and synchronicity. The solution consists of A sending B a message (i.e., a request) and B sending back a message (i.e., a reply) after processing the request, as you can see in Fig. 2. For A, this simple mechanism provides a modest level of reliability and synchronization, because the reply proves that B received the request and can provide relevant information about B's state. Furthermore, if A does not receive a reply within a specific amount of time (i.e., a timeout), it can resend the request. It can continue to timeout and retry until it eventually receives a reply or it exceeds some maximum number of retries. This "timeout/retry" behavior is the essence of the request-reply pattern.

TABLE I. COMMDP PATTERNS

Name	Consequences			
	R	S	L	A
Request-Reply	2	2	1	1
Request-Reply-Acknowledge	3	3	1	1
Idempotent Retry	3	1	1	1
Intermediate State Messages	3	3	3	1
Second Channel	1	3	3	1
Front End	1	1	1	3
Proxy	1	1	1	3
Reliable Multicast	3	3	1	2
Publish-Subscribe	2	1	1	3

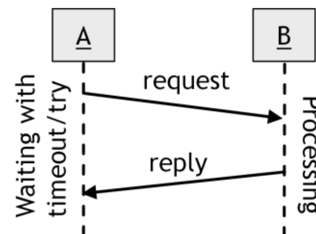

Figure 2. Request-Reply Message Sequence

The *Request-Reply-Acknowledge* pattern extends this solution with a third message (an acknowledgement) that A sends to B after receiving the reply, and gives B a timeout/retry behavior with respect to its sending of the reply and waiting for an acknowledgement, see Fig. 3. This pattern is useful in situations were significant processing may occur on A after receiving the reply or when it is problematic for B to reprocess duplicated requests caused by A's timeout/retry behavior. With this pattern, instead of reprocessing a duplicate request, B can simply cache its replies and resends them to A when necessary. The acknowledgement tells B that A has received the reply and, thus, can remove it from its cache. This pattern offers more reliability and synchronization than request-reply, but at the cost of an additional message.

Copyright (c) IARIA, 2016.

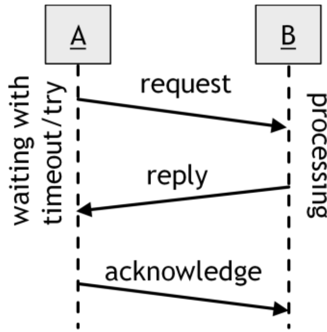

Figure 3. Request-Reply-Acknowledge Message Sequence

The *Idempotent Retry* pattern [46] captures a different solution to the problem of processing duplicate requests. Like Request-Reply, its solution consists of A sending a request to B with a timeout/retry behavior and B sending a reply back to A. But, unlike Request-Reply, the semantics of the protocol dedicate the processing of the request must be idempotent. This pattern applies to situations where the requested processing is relatively light, i.e., less expensive than caching replies.

The next pattern, *Intermediate State Message*, is also similar to Request-Reply, but addresses the problem of long-running conversations due to request actions taking substantial amounts of time to complete. To solve this problem, it has B send A one or more intermediate messages that reflect its current state. For example, B may send a message immediately after receiving the request to let A know that it got the request, another message when the processing is 10%, another at 20% complete, and so on. Each intermediate message provides state information about B, which improves synchronization in the presence of time-consuming actions, see Fig. 4.

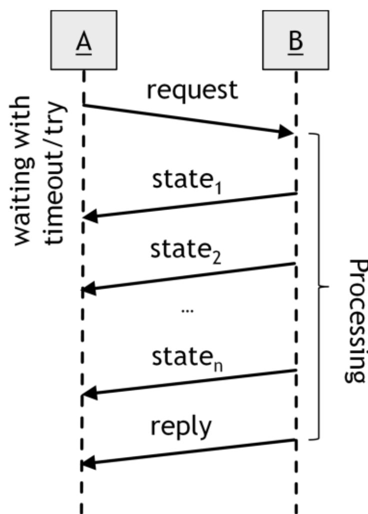

Figure 4. Intermediate State Message Sequence

The *Second Channel* pattern is also for situations involving long-running conversations, but ones dominated by significant amounts of data transfers instead of time-consuming actions. Because the large data transfers can delay intermediate state messages, this pattern's solution suggests opening a second communication channel between A and B that is dedicated to data transfer, leaving the original communication channel available for intermediate state or control messages, as it is shown in Fig. 5. The File Transfer Protocol (FTP) and its variations are classical examples of this pattern[10][45].

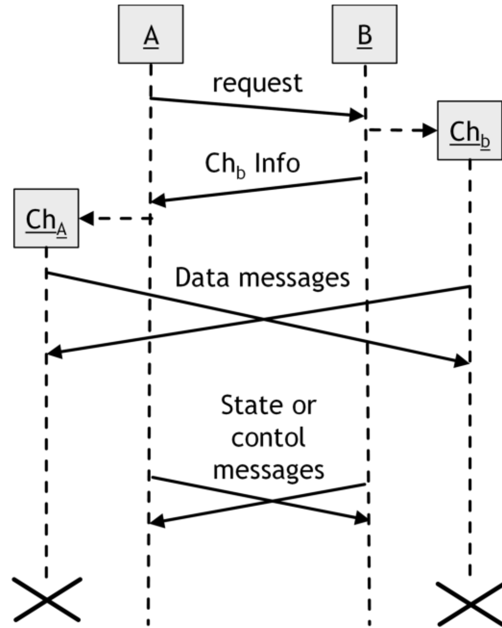

Figure 5. Second Channel Message Sequence

The *Front End* pattern addresses the problems of making the location of shared resource transparent to the client, allowing the number of resources to change dynamically. It has a resource client send requests to a front-end process that automatically redistributes them to appropriate resource managers, B processes. After processing the request, a resource manager replies back to the client directly, for a graphic description of this pattern, you can see Fig. 6. The front-end process can use a variety of criteria to decide how to redistribute requests, including request type, resource type or identity, and resource manager load. By itself, this pattern's

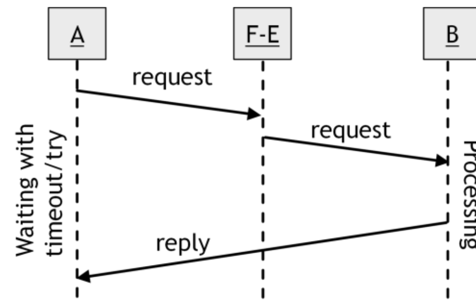

Figure 6. Front End Message Sequence

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-498-5

primary focus is on the distribution and scalability of resources.

Like the Front End, the *Proxy* pattern, presented in Fig. 7, introduces a process between a resource client and a resource manager. However, the intermediate process, called a proxy, serves other functional purposes besides re-distribution of the requests, for example it may provide authentication, access control, audit logging, and data transformation functionality. Also, the resource manager returns replies through the proxy to client, completely isolating the client from the resource manager.

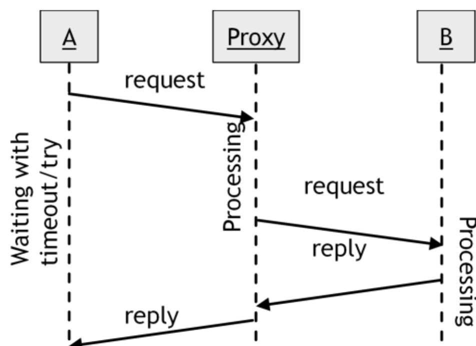

Figure 7. Proxy Message Sequence

The *Reliable Multicast* pattern builds on the multicast idiom to provide reliability and synchronization among a group of processes. Its solution is a protocol that starts with a process A sending a request message to a group of process, B={b_1, .., b_n}. Each process b_i sends a reply back to A when it receives the request and is ready to process it. After A receives reply from all B processes, then A will multicast a go-ahead message back out to all B message indicating that they can proceed with the processing of the request, shown in Fig. 8. In this way, the execution of the request is synchronized among all of the B processes. If A fails to receive a reply from every B process, it can resend the request to some or all of them until it gets a reply from all of them or terminates the conversation

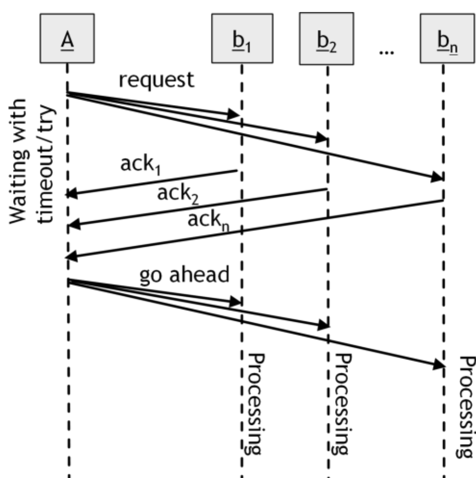

Figure 8. Reliable Multicast Message Sequence

as failed. This pattern focuses on providing strong reliability and synchronization, but can also help with scalable distribution of resources.

Finally, the *Publish-Subscribe* [8] pattern is a powerful mechanism for decoupling message senders (publisher) from message receivers (subscribers). With this pattern, an intermediate process acts as a store-and-forward buffer for message transmission with the capabilities for managing subscribers and delivering individual message to multiple subscribers.

C. *CommDP: Pattern Relationships and Composition*

Patterns are rarely used in isolation; instead, developers combine their solutions to solve complex problems. Virtually any of the CommDP patterns could be combined with any other pattern, but the more useful combinations are ones that have complimentary characteristics, like Request-Reply with Second Data Channel or Request-Reply Acknowledge with Front End.

To ensure that the CommDP pattern set was as minimal as possible, we did not include in any pattern in CommDP that was simply an aggregation of two or more patterns. For example, there is a common type of distributed system that deals with information flow and processing. In such systems, a process A might send a request to B through a series of intermediate proxy-like processes that transform or augment data in request on its way to B. At each intermediate step, a reply is sent back to A, informing it of the message's process. Eventually, when the transformed message arrives at B and processes it, then B sends a final reply message back to A. This particular solution offers good reliability, synchronization, longevity, and adaptability to scalable distribution, but it is actually just a composition of the Proxy pattern (applied perhaps multiple times) and the Intermediate State Message pattern.

VII. SUMMARY AND FUTURE WORK

CommDP pulls together reusable solutions to reoccurring design problems with ACPs, filling a much needed gap in the knowledge base for developers of distributed systems. We have characterized the nature of the problems that the CommDP patterns address and the consequences of their solution in terms of four desirable qualities, namely: reliability, synchronicity, longevity, and adaptability for scalable distribution. These qualities are both instructive and discriminating, in that they can help a developer understand the solutions and choose the most appropriate solution for a given situation. However, more work needs to be done to formalize these qualities and to solidify their sufficiently and completeness relative communication-protocol design. So this is one of our research group's immediate goals.

We also hope to investigate other qualities, like cohesion and modularization that might be valuable for protocol design even if they are not good discriminators for design patterns. Being able to reason about assess, and teach these qualities more formally will help developers create better distributed systems.

Finally, over time, we hope the expand the patterns in CommDP, without adding any that are just compositions of

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-498-5

existing patterns, to encompasses a boarder range of reusable solutions for ACPs.

REFERENCES

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5 edition. Boston: Pearson, 2011.

[2] "Distributed computing," *Wikipedia, the free encyclopedia*. 27-Feb-2016.

[3] S. W. Clyde, "Object mitosis: a systematic approach to splitting objects across subsystems," in *, Proceedings of the Third International Workshop on Object Orientation in Operating Systems, 1993*, 1993, pp. 182–185.

[4] "Communications protocol," *Wikipedia, the free encyclopedia*. 10-Apr-2016.

[5] "protocol | computer science," *Encyclopedia Britannica*. [Online]. Available: http://www.britannica.com/technology/protocol-computer-science. [Accessed: 20-Apr-2016].

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 edition. Addison-Wesley Professional, 1994.

[7] J. O. Coplien and N. B. Harrison, *Organizational Patterns of Agile Software Development*. Upper Saddle River, NJ: Prentice Hall, 2004.

[8] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*, Volume 4 edition. Wiley, 2007.

[9] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977.

[10] C. White, *Data Communications and Computer Networks: A Business User's Approach*, 7 edition. Boston, MA: Cengage Learning, 2012.

[11] "ISO/IEC 10026-1:1992 - Information technology -- Open Systems Interconnection -- Distributed Transaction Processing -- Part 1: OSI TP Model." [Online]. Available: http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=17979. [Accessed: 20-Apr-2016].

[12] K. Beck and W. Cunningham, "Using Pattern Languages for Object-Oriented Programs," in *Object-Oriented Programming, Systems, Languages, and Application*, Sep. 1987.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*, Volume 1 edition. Chichester ; New York: Wiley, 1996.

[14] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*, Volume 2 edition. Chichester England ; New York: Wiley, 2000.

[15] J. Tidwell, *Designing Interfaces*, 2 edition. Sebastopol, CA: O'Reilly Media, 2011.

[16] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching," in *Pattern Languages of Program Design*, New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 529-545.

[17] I. Pyarali, T. Harrison, and D. Schmidt, "Asynchronous Completion Token: an Object Behavioral Pattern for Efficient Asynchronous Event Handling," in *Proc. 3rd Annual Conference on The Pattern Languages Programs*, 1997, pp. 1-7.

[18] I. Pyarali, T. Harrison, D. C. Schmidt, and T. D. Jordan, "Proactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events," in *Pattern Languages of Program Design (J. O. Coplien and D. C. Schmidt, eds.)*, Reading, MA: Addison-Wesley, 1995.

[19] M. K. Douglas C. Schmidt, "Leader/Followers - A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching," in *7th Pattern Languages of Programs Conference*, Allerton Park, Illinois, 2000.

[20] D. C. Schmidt, "Strategized locking, thread-safe interface, and scoped locking," *C Rep.*, vol. 11, no. 9, 1999.

[21] R. G. Lavender and D. C. Schmidt, "Active Object an Object Behavioral Pattern for Concurrent Programming" in *Pattern Languages of Program Design 2 edited by John Vlissides, Jim Coplien, and Norm Kerth.*, Boston, MA: Addison-Wesley, 1996.

[22] D. C. Schmidt, "Monitor Object," in *Pattern-Oriented Software Architecture (F. Buschmann, K. Henney, D. C. Schmidt)*, vol. 4, West Sussex PO19 8SQ, England: John Wiley & Sons Ltd, 2007, pp. 368-369.

[23] D. C. Schmidt and C. D. Cranor, "Half-Sync/Half-Async," presented at the Second Pattern Languages of Programs, Monticello, Illinois, 1995.

[24] D. C. Schmidt, N. Pryce, and T. H. Harrison, "Thread-Specific Storage for C/C+," *More C Gems*, vol. 17, p. 337, 2000.

[25] D. C. Schmidt and T. Harrison, "Double-checked locking," in *Pattern languages of program design*, vol. 3, 1997, pp. 363–375.

[26] L. Rising, *Design Patterns in Communications Software*, 1 edition. Cambridge ; New York: Cambridge University Press, 2001.

[27] P. Jain and D. C. Schmidt, "Service Configurator: A Pattern for Dynamic Configuration of Services," in *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, Berkeley, CA, USA, 1997, pp. 16–16.

[28] R. C. Martin, D. Riehle, and F. Buschmann, *Pattern Languages of Program Design 3*, 1 edition. Reading, Mass: Addison-Wesley Professional, 1997.
[29] L. Rising, *Design Patterns in Communications Software*, 1 edition. Cambridge ; New York: Cambridge University Press, 2001.
[30] D. C. Schmidt, “Using design patterns to develop reusable object-oriented communication software,” *Commun. ACM*, vol. 38, no. 10, pp. 65–74, 1995.
[31] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2011.
[32] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer Science & Business Media, 2014.
[33] “Patterns for Distributed Real-time and Embedded Systems.” [Online]. Available: https://www.dre.vanderbilt.edu/~schmidt/patterns-ace.html. [Accessed: 04-Mar-2016].
[34] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1 edition. Addison-Wesley Professional, 1994.
[35] E. Yourdon and L. L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1979.
[36] “Modularity,” *Wikipedia, the free encyclopedia*. 11-Apr-2016.
[37] M. Burrows, “The Chubby Lock Service for Loosely-coupled Distributed Systems,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2006, pp. 335–350.
[38] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Commun ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[39] M. Raynal, “About Logical Clocks for Distributed Systems,” *SIGOPS Oper Syst Rev*, vol. 26, no. 1, pp. 41–48, Jan. 1992.
[40] F. Mattern, “Virtual time and global states of distributed systems,” in *Parallel and Distributed Algorithms*, 1989, pp. 215–226.
[41] C. Fidge, “Logical Time in Distributed Computing Systems,” *Computer*, vol. 24, no. 8, pp. 28–33, Aug. 1991.
[42] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5 edition. Boston: Pearson, 2011.
[43] H. T. Kung and J. T. Robinson, “On Optimistic Methods for Concurrency Control,” *ACM Trans Database Syst*, vol. 6, no. 2, pp. 213–226, Jun. 1981.
[44] M. Voelter, “Patterns for Handling Cross-cutting Concerns in Model-Driven Software Development,” in *ResearchGate*, 2005.
[45] A. S. Tanenbaum and D. Wetherall, *Computer networks*, 5th ed. Boston: Pearson Prentice Hall, 2011.
[46] R. Daigneau, *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*, 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2011.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-498-5