GreatFree as a Generic Distributed Programming Language and the Foundation of the Cloud-Side Operating System

Bing Li
GreatFree Research Lab
Jiangsu, China
E-mail: bing.li@asu.edu

***Abstract*—GreatFree is a generic distributed programming language to develop various distributed systems over the Internet-oriented computing environment. The fundamental characters of GreatFree are shaped by three essential techniques, including the message-passing, the physical-machine-visible, and the thread-visible. More important, GreatFree is equipped with three additional distinguished mechanisms, i.e., the distributed primitives, the distributed common patterns, and the distributed threads on the application level, which are sufficient to turn GreatFree into a generic distributed programming technology. To the best of our knowledge, compared with any others, GreatFree is the first one to achieve the goal. Thereafter, GreatFree is capable of exploiting distributed computing resources flexibly to adapt to any heterogeneous environments with a uniform solution. It indicates that GreatFree represents the common principles existed in various complicated computing circumstances over the Internet. That inspires that GreatFree is a proper technology to build a new concept of cloud computing environment, i.e., the cloud-side operating system, which dominates diverse distributed computing resources upon the common principles of GreatFree. Such a system is a generic development and running environment for any distributed systems. Without doubt, within the environment, GreatFree is the unique choice to program any distributed systems in a scalable manner.**

Keywords- Cloud-side Operating System; Generic Distributed Programming Languag; Distributed Primitives; Application-level Threading on Messaging; the Distributed Common Patterns

I. INTRODUCTION

GreatFree is a generic distributed programming language for the Internet-oriented computing environment. It has the three necessary characters to become such a technique, i.e., the message-passing between threads, the threading programmable, and the physical distributed node programmable. More important, three additional distinguished characters are sufficient to support GreatFree to become a generic and rapid distributed programming paradigm. Those characters consist of the distributed primitives, the distributed common patterns, and the application-level threading on messaging. With the emergence of GreatFree, it inspires the generation of the new concept of distributed development and running environment, i.e., the cloud-side operating system.

GreatFree is a generic programming paradigm for the Internet-oriented computing environment. Three technologies, i.e., the Distributed Primitives (DP), the Application-level Threading on Messaging (ATM), and the Distributed Common Patterns (DCP), are proposed to form the distinguished characters of GreatFree. The DP consists of a series of the distributed primitive application programming interfaces. Any distributed components and systems are originated from the DP through self-derivation without the support of any third-party distributed techniques. The ATM is the distributed, application-level, and asynchronous message-passing threading, which aims to implement the most fine-grained distributed concurrent systems to leverage computing resources in various distributed environments conveniently. The DCP unveils that the code of any heterogeneous distributed systems is constructed with a limited number of the common code-level design patterns through self-derivation. With the support of those techniques,

GreatFree not only becomes a generic and rapid paradigm of distributed programming but also establishes the basis of the first distributed programming language for the Internet-oriented computing environment.

The DP represents the most fundamental programming components for the development of various distributed systems. As one of the foundations of GreatFree, the DP is the basis of general-purpose techniques for distributed systems development. It shapes GreatFree to become a full coverage and rapid development technique in a sense that it not only hides tedious details but also exposes indispensable elements. The DP is equivalent to the most basic and mandatory distributed computing resources and mechanisms to construct any distributed systems. If any element of the DP is missed, it definitely results in the failure of programming with GreatFree to implement any systems. On the other hand, if any of the ones encapsulated by the DP is exposed, it raises the programming efforts and lowers the quality of developed systems obviously.

Different from other concurrent mechanisms [1~15] for distributed programming, the ATM threads are visible to developers, i.e., the threads can be manipulated by passing application-dependent messages without worrying any native characters of threads. Because of the complexity of the Internet-oriented computing environment, it is impossible to predefine an omni-potent concurrency instrument, which not only conceals every detail of threads for rapid programming but also adapts to sophisticated cases. Hence, it is required to allow developers themselves to create, monitor, reuse, and collect threads directly on the application level. Thus, the support of visible threads is necessary for a generic distributed programming paradigm. In addition, to isolate developers from the tedious synchronization workload of threading, the ATM is founded on the basis of the message-passing rather than that of the memory-sharing in traditional methodologies [16~22]. Using the ATM, developers are able to handle any number of remote threads on arbitrary distributed nodes to process intricate distributed tasks concurrently through message-passing only. Consequently, a single ATM thread is equivalent to a distributed node accomplishing scheduled tasks in a serial way such that developers can program with the ATM threads in the same way as distributed nodes to construct various complicated distributed systems in a higher quality. As a novel distributed concurrency technique, the ATM is another crucial foundation of GreatFree to become a generic distributed programming paradigm.

The DCP is a phenomenon existing natively in any distributed system instead of a contrived technique. It is evident that the DCP exists pervasively in all the code of distributed programs. With respect to the large amount of distributed programming experiences in various environments, it discovers that the code of heterogeneous distributed systems abides by a limited number of the homogeneous code-level design patterns. Although it is always necessary to derive diverse DAAs to raise the rapidness of programming complicated distributed systems, the types of code-level design patterns do not change with the new proposed APIs. Only the DCP is sufficient to adapt to any scenarios since the patterns for DAA are always the straightforward aggregations of the DCP and nothing else needs to be invented for DAA. Thus, when programming with DAA, the same code structures originated from the DCP are reusable for distinct distributed systems. The phenomenon reveals the truth that the code structures of a distributed system are independent of its distributed natures.

II. RELATED WORK

Distributed programming is an evergreen topic such that it contains plentiful solutions. According to their originally target computing environments by default, all of them are classified as the Sequential and Standalone Paradigm (SSP), the Distributed Frameworks Paradigm (DFP), and the Distributed Programming Paradigm (DPP). The DPP, the primary methodology GreatFree competes with, contains a variety of mutations aiming to become general-purpose solutions.

A. The Sequential and Standalone Paradigm

The SSP specifies the programming methodologies that implement a system running in the sequential and standalone manner by default. Most traditional high-level programming

languages [23~34] belong to the category. When the SSP was invented, the primary effort was focused on replacing the machine-dependent code with the nature-language-like syntax and semantics. They do not take into account the issues of the concurrency and distribution. With the rapid development of computing technologies, it is required to program concurrent and distributed systems using those sequential and standalone languages. For that, the techniques of threading and networking are proposed to support the SSP programs to be executed concurrently in a network environment. When programming with those techniques, developers are required to transform the sequential and standalone instructions to the concurrent and distributed ones with those attached techniques. The procedure is notoriously difficult such that even proficient developers usually avoid doing that if alternative solutions are available.

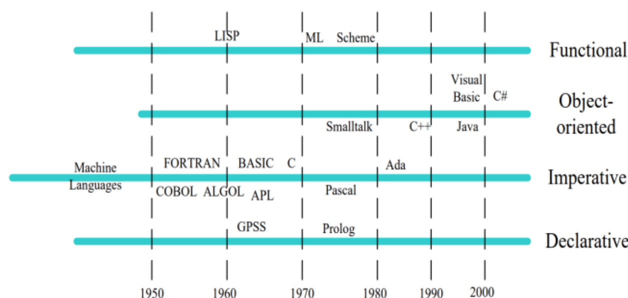

Figure 1. Sequential and Standalone Paradigm

Simply put, to program a server, i.e., one single physical distributed node, with the SSP, the effort is intolerable although such a server is the simplest component within various distributed systems. The programming efforts for the server include networking, serialization, message-passing, message scheduling, threading management, threading synchronization, threading scheduling, resource management and so forth. Additionally, the solutions to those issues always change for specific applications, such as lightweight or heavyweight, streaming or messaging, idle or busy, centralized or decentralized, stable or unstable, heterogeneous or homogeneous, machined or socialized, and so forth. Even though all the issues are resolved, it is still tough to extend them to implement a scalable large-scale system since the workload is always raised exponentially. Besides, the incompatible code structures also reflect the difficulty of programming distributed systems with the traditional languages. Even the same developer programs the same server with different code patterns if lacking for experiences and references. It brings forth the difficulty to manage, reuse and debug for further development and collaboration.

B. The Distributed Frameworks Paradigm

Since it is tough to implement distributed systems with the SSP, as the semi-constructed systems, the distributed frameworks are employed to simplify the development in most cases. Because the DFP resolves all the distributed issues and make them invisible in one specific domain, developers are able to work within a virtualized computing environment in which no concurrent and distributed issues need to be considered. Then, they are concentrated on programming upper level applications in a sequential and standalone manner. This approach is the most rapid such that it becomes popular nowadays.

However, the DFP is never a generic solution for the complexity of distributed computing environments. Instead, all the DFP solutions [35~60] are application-specific such that it hides developers from all the distributed issues for one particular scenario in the enterprise-level distributed computing environment. The current existing distributed frameworks cover the issues such as the distributed objects environment, the remote procedure call, the map/reduce concurrency, the clustering, the infrastructure for the enterprise environment, the data management, the streaming, the high-level scripting, and the customized applications. Unfortunately, it is impossible to establish a framework to make all the distributed issues transparent in the Internet-oriented computing environment. If one particular application is suitable to one of those distributed frameworks luckily, it results in low development efforts. If not, there is no way to make changes on those frameworks to adapt to specific requirements. A common case is that a bunch of distributed frameworks have to be accumulated in one specific application to fulfill respective scenarios. Such a system is always cumbersome in terms of management and resources consuming. Therefore, the DFP is far from perfect since the software

development is degenerated from straightforward programming with a single full-fledged language to patching, scripting, configuring, or integrating multiple heterogeneous frameworks.

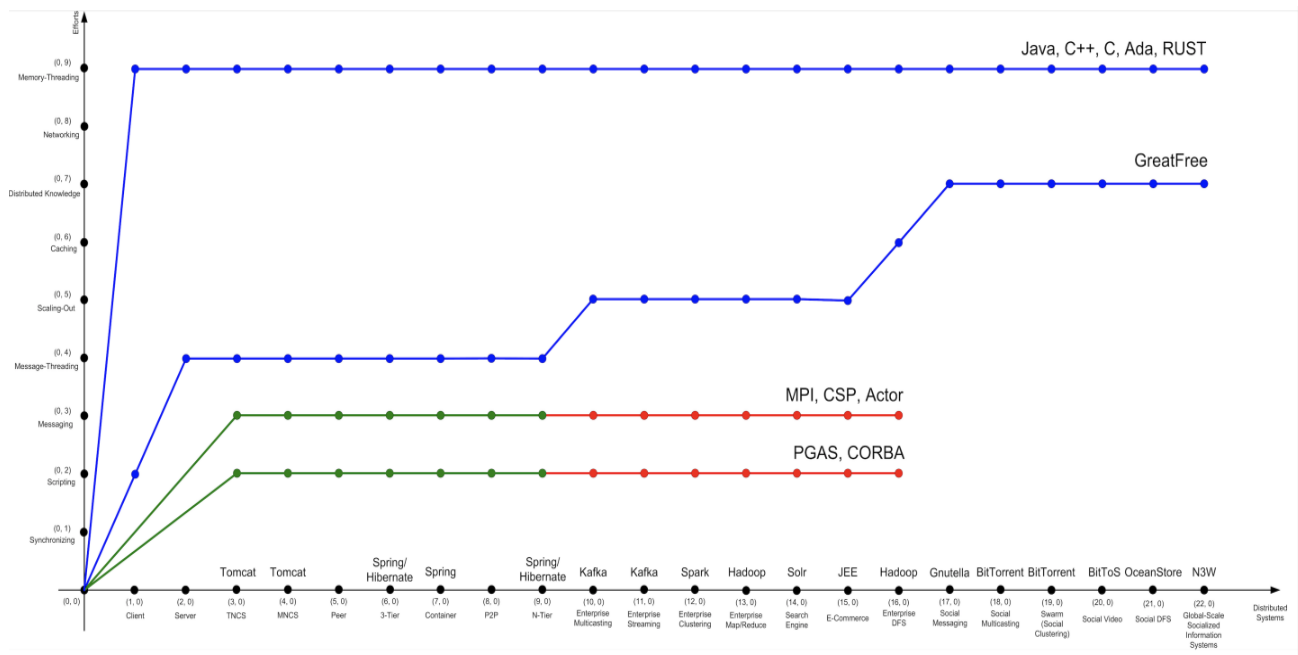

Figure 2. Distributed Frameworks Paradigm

C. The Distributed Programming Paradigm

The Distributed Programming Paradigm (DPP) [61~82] is defined as the methodology that aims to develop the systems in the computing environment in which multiple computers are connected through networking.

For the complexity of distributed computing environments, there are numerous mutations in the DPP. As any computing systems perform behaviors to manipulate data, it is appropriate to identify them upon their approaches of accessing and exchanging distributed data among distributed threads and processes. According to that, the DPP is categorized into the Memory-Sharing Paradigm (MSP) and the Message-Passing Paradigm (MPP). The MSP attempts to create a virtualized uniform memory space for a distributed computing environment. Hence, network locations are invisible, and data is retained in a unique memory space from a programmer's point of view. Because of that, a distributed computing environment is transformed to a standalone one. With the support of the MSP, it is unnecessary to take care of any distributed techniques to implement distributed systems. On the other hand, the MPP believes it is feasible for simple scenarios to construct such a homogeneous memory space. For complicated cases, it is impossible. Even though for those simple ones, it causes additional problems, such as heavy synchronization, low performance, and low scalability. Therefore, the MPP claims that multiple independent memory spaces are the foundation to process distributed data. To establish asynchronous, high performance, and scalable distributed systems, instead of sharing, data is passed as messages among distributed entities, including threads and processes, within isolated memory spaces.

In addition, as one of the most important components to program distributed systems, the concurrency implementation is another proper indicator to differentiate various paradigms. In accordance with the visibility of threading, the DPP is divided into the Threading Invisible Paradigm (TIP) and the Threading Visible Paradigm (TVP). Because of the difficulty to program with traditional threading, all the existing paradigms encounter the dilemma, i.e., they have to make a single choice between the adaptability to various scenarios and the rapidness of programming. Each of them either loses the adaptability to gain the rapidness or abandons the

rapidness to obtain the adaptability. None of them wins both of them. The TIP hides threading to lower the difficulty to concurrency programming for distributed systems. That is the choice of most paradigms such that it proves the toughness of threading further. To do that, a concurrency pooling mechanism needs to be predefined to manage threads running asynchronously. Different from the TIP, the TVP exposes threading to adapt to various scenarios since it is impossible to create an omnipotent thread management mechanism to deal with unpredictable cases. For complicated systems, it is required for programmers to design the specific pooling for threading based on the certain domain knowledge. Therefore, the TVP declares that visible threading is a mandatory condition to accommodate to various contexts.

Finally, any distributed systems are constructed upon multiple computing devices. Thus, it is necessary to convert such a standalone device to a distributed node, which is able to interact with others. The technique of conversion is called the distributed modeling. To speed up the development of distributed systems, many variants of the DPP hide the modeling from programmers. They intend to create an abstract object to replace the physical heterogeneous distributed node. Programming with the logical entities instead of physical nodes, the tedious details of distributed environments are filtered out such that the efforts are focused on composing those homogeneous components. Then, the programming rapidness is raised obviously. Such a paradigm is called the Modeling Invisible Paradigm (MIP). However, because of the complexity of distributed computing environments, hiding physical distributed nodes results in the fact that programmers lose the possibility to access locations of distributed nodes, organize distributed nodes into one particular topology, and establish effective interactions among distributed nodes. Those issues are critical for a complicated distributed system and it is impossible to predefine them without taking into account requirements in one specific circumstance founded on those distributed nodes. For that, another approach, the Modeling Visible Paradigm (MVP) is proposed to overcome the drawbacks of the MIP.

On the other hand, all the paradigms can be classified roughly into the high-level one and the low-level one as well. The high-level paradigm strives to construct a logical prototype that is as independent of the physical distributed computing environments as possible to raise the rapidness of distributed programming. On the contrary, the low-level one aims to be closed to the physical distributed computing environments such that it is possible to guarantee the generality of distributed programming. With respect to the principle, the high-level one consists of the MSP, the TIP, and the MIP whereas the low-level one includes the MPP, the TVP, and the MVP. Moreover, among those approaches, data accessing determines others to a large extent since the functions of a computing system can be summarized as reading or writing data. For that, the DPP is mainly classified as the MSP and the MPP, which are the high-level and the low-level respectively. It is unreasonable to introduce the TVP and the MVP, which are low-level techniques compared with the TIP and the MIP, into the MSP since the paradigm conceals all the distributed details. Different from the MSP, as a low-level paradigm, the MPP is open enough to play the role of technical basis such that high-level ones are allowed to be established on it and low-level ones are employed to raise its generality.

TABLE I. THE CATEGORIZATIONS OF THE PPP INSTANCES

ID	Technique	MSP	MPP	TIP	TVP	Year of Birth
1	Id	Y	N	Y	N	1975
2	Sisal	Y	N	N	Y	1983
3	Occam	N	Y	N	Y	1983
4	Multilisp	Y	N	Y	N	1985
5	Newsqueak	N	Y	N	Y	1985
6	ParLog	Y	N	Y	N	1987
7	C*	Y	N	Y	N	1987
8	Joyce	N	Y	N	Y	1987
9	SequenceL	Y	N	Y	N	1989
10	Charm++	N	Y	Y	N	1989
11	Lustre	Y	N	Y	N	1991
12	HPF	Y	N	Y	N	1991
13	Alef	N	Y	N	Y	1992
14	ZPL	Y	N	Y	N	1993
15	SuperPascal	N	Y	N	Y	1993
16	OpenMP	Y	N	Y	N	1997
17	Titanium	Y	N	Y	N	1998
18	UPC	Y	N	Y	N	1999
19	BMDFM	Y	N	Y	N	2002
20	CnC	Y	N	Y	N	2004
21	XC	N	Y	N	Y	2005
22	Fortress	Y	N	N	Y	2006
23	Sequoia++	Y	N	Y	N	2006
24	Preesm	N	Y	Y	N	2008
25	Chapel	Y	N	Y	N	2009
26	C++AMP	Y	N	Y	N	2011

In addition to those classic ones, as one subset of the DPP, the Parallel Programming Paradigm (PPP) can be regarded as an early version to a special distributed computing environment. The PPP provides an abstract prototype for concurrent executions to attain high performance on a single standalone physical computer equipped with multiprocessors. Similar to the DPP, the PPP consists of the MSP, the MPP, the TIP, and the TVP as well. Neither MIP nor the MVP is associated with the PPP because the PPP supports the standalone computing device only. Compared with others of the DPP, the computing environment of the PPP is highly homogeneous since those multiprocessors within a computer are identical and each of them has an equivalent assignment of computer resources and capabilities such that there are no differences among those processors when exploiting them to accomplish multiple tasks concurrently. Therefore, it is easy to design highly abstract programming components to conceal low-level details. Because of that, most variants of the PPP are classified as the MSP and the TIP.

TABLE II. THE SUMMARY OF THE PPP

Paradigm	Proportion				
	26(100%)	1970s	1980s	1990s	2000s
MSP	18(69%)	1(4%)	5(19%)	6(23%)	6(23%)
MPP	8(31%)	0(0%)	4(15%)	2(8%)	2(8%)
TIP	18(69%)	1(4%)	5(19%)	6(23%)	6(23%)
TVP	8(31%)	0(0%)	4(15%)	2(8%)	2(8%)
MSP & TIP	16(62%)	1(4%)	4(15%)	6(23%)	5(19%)
MSP & TVP	2(8%)	0(0%)	1(4%)	0(0%)	1(4%)
MPP & TIP	2(8%)	0(0%)	1(4%)	0(0%)	1(4%)
MPP & TVP	6(23%)	0(0%)	3(12%)	2(8%)	1(4%)

According to the above discussions, as a typical methodology of the DPP, GreatFree is categorized into the MPP, the TVP, and the MVP. Aiming to be a generic paradigm, for the issues of data accessing, threading, and modeling, GreatFree always chooses the low-level solution rather than the high-level one. In other words, GreatFree has to propose distinct resolutions to avoid the inefficiency of programming. To break out the dilemma, GreatFree possesses the two distinguished characters, including the DP as the primitive distributed programming components and the DCP as the common homogeneous distributed code structures, to programming distributed systems rapidly. Moreover, it puts forward the distinct solution, the ATM, to the tough issue of threading. Therefore, GreatFree not only simplifies distributed programming mechanisms in the way to conceal those intricate techniques but also abstracts distributed resources and technical details to a degree to sustain the sufficient adaptability to various distributed computing environments.

III. GREATFREE AS A GENERIC DISTRIBUTED PROGRAMMING LANGUAGE

GreatFree is a generic distributed programming paradigm in the Internet-oriented computing environment. The three fundamental techniques, i.e., the Distributed Primitives (DP), the Application-level Threading on Messaging (ATM), and the Distributed Common Patterns (DCP), are proposed to achieve the goal to be a generic paradigm in the highly heterogeneous distributed computing circumstance.

The DP is the most fundamental elements that are sufficient and necessary to program any distributed systems. The ATM is a distributed concurrency programming technique distinguished from others by the mechanisms of the message-passing and the visible-threading on the application level. The DCP is a rapid distributed programming solution on condition that any heterogeneous distributed systems can be constructed with the common homogeneous code-level design patterns.

In general, GreatFree is the paradigm of the MPP, the TVP, and the MVP in the domain of distributed programming. Moreover, GreatFree is distinct from any others in the discoveries of the rudimentary and universal programming components, the application-level fine-grained concurrency model, and the homogeneous code structures. Thus, GreatFree becomes a new paradigm in the fashion of being individual-respected, messaging-oriented, threading-visible, and self-derivable such that it becomes unique as a generic distributed programming methodology in the heterogeneous computing environment of the Internet.

TABLE III. THE CATEGORIZATIONS OF THE MSP AND MPP INSTANCES

ID	Technique	MSP	MPP	TIP	TVP	Year of Birth
1	CSP	N	Y	N	Y	1978
2	Ada	Y	N	Y	N	1980
3	Emerald	Y	N	Y	N	1985
4	Linda	Y	N	Y	N	1986
5	Erlang	N	Y	Y	N	1986
6	LabVIEW	N	Y	Y	N	1986
7	Hermes	N	Y	N	Y	1986
8	SR	Y	N	Y	N	1988
9	Concurrent Smalltalk-90	N	Y	Y	N	1989
10	Haskell	Y	N	Y	N	1990
11	Janus	N	Y	Y	N	1990
12	CORBA	Y	N	Y	N	1991
13	MPI	N	Y	Y	N	1991
14	Oz	N	Y	N	Y	1991
15	SHMEM	Y	N	Y	N	1993
16	CML	N	Y	N	Y	1993
17	Glenda	Y	N	Y	N	1994
18	Limbo	N	Y	Y	N	1995
19	Millepede	Y	N	Y	N	1996
20	Joule	N	Y	Y	N	1996
21	E	Y	N	Y	N	1997
22	MPJ	N	Y	N	Y	1999
23	MPD	Y	N	Y	N	2000
24	SALSA	N	Y	Y	N	2001
25	CAL	N	Y	Y	N	2001
26	D	N	Y	N	Y	2001
27	X10	Y	N	Y	N	2004
28	JoCaml	N	Y	N	Y	2004
29	JCSP	N	Y	N	Y	2005
30	PyCSP	N	Y	N	Y	2006
31	Akka	N	Y	Y	N	2009
32	Go	N	Y	Y	N	2009
33	Axum	N	Y	Y	N	2009
34	Bloom	Y	N	Y	N	2010
35	Rust	N	Y	N	Y	2010
36	Ateji PX	N	Y	Y	N	2010
37	Elixir	N	Y	Y	N	2011
38	Julia	N	Y	N	Y	2012
39	Akka.NET	N	Y	Y	N	2013

TABLE IV. THE SUMMARY OF THE MSP AND THE MPP

Paradigm	Proportion				
	39(100%)	1970s	1980s	1990s	2000s
MSP	13(33%)	0(0%)	4(10%)	6(15%)	3(8%)
MPP	26(67%)	1(3%)	4(10%)	7(18%)	14(36%)
TIP	28(72%)	0(0%)	7(18%)	26(23%)	11(28%)
TVP	11(28%)	1(3%)	1(3%)	3(8%)	6(15%)
MSP & TIP	13(33%)	0(0%)	4(10%)	6(15%)	3(8%)
MSP & TVP	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)
MPP & TIP	15(39%)	0(0%)	4(10%)	3(8%)	8(21%)
MPP & TVP	11(28%)	1(3%)	1(3%)	3(8%)	6(15%)

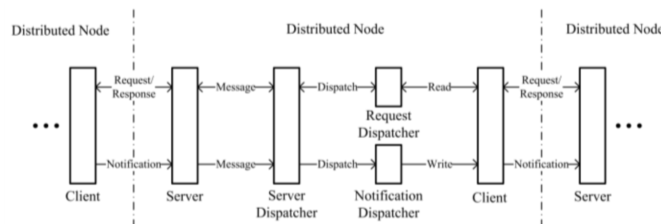

Figure 3. GreatFree Paradigm - DP

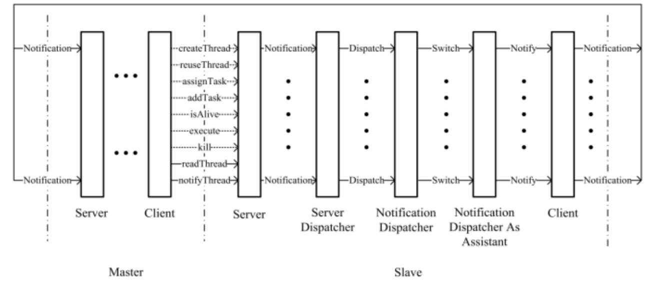

Figure 4. GreatFree Paradigm - AMTL

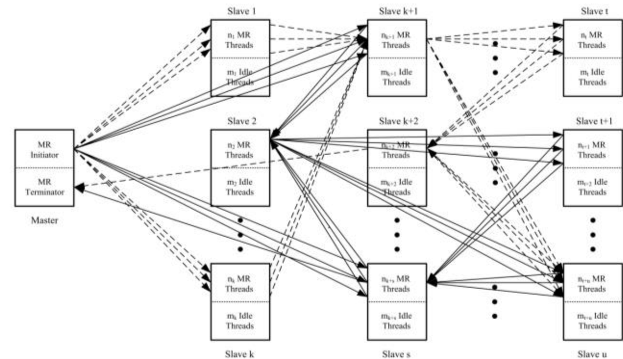

Figure 5. GreatFree Paradigm – AMTL for Map/Reduce

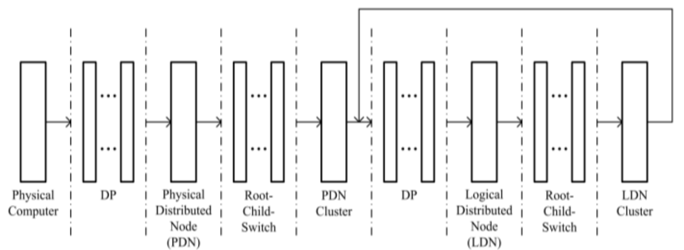

Figure 6. GreatFree Paradigm - SPRA

A. The DP

The DP represents the most fundamental distributed elements, which are sufficient and necessary to program any distributed systems in the Internet-oriented computing environment. The DP is the foundation of GreatFree to be a generic programming paradigm. The DP is made up of a series of the distributed primitive APIs. Programming with the DP only, it is rapid to create various distributed systems, such as the simplest ones, the distributed advanced APIs, and the distributed frameworks, in any environments. Even the most complicated one, the global scale socialized heterogeneous information system over the Internet, can be programmed with the DP. It demonstrates that the DP is also the basis of GreatFree as the self-derivable programming paradigm.

1) The Distributed APIs

The DP is the most elementary application programming interfaces to accomplish the intrinsic distributed functionalities. It consists of the three subsets, i.e., the distributed modeling, the distributed messaging, and the distributed dispatching. The distributed modeling transforms a single standalone physical computing device to one physical distributed node, i.e., one physical client or one physical server, such that the device can interact with any others within the Internet environment. The distributed messaging describes the interactions of requesting or eventing via the messages in the plain object-oriented form transmitted over the Internet. The distributed dispatching processes incoming messages concurrently in a scaling-up manner on a distributed node. As the most complicated component in the DP, the distributed dispatching includes the message dispatcher, the messaging thread pools, and the messaging threads. After a long-term experimenting, all of them are polished carefully to keep the balance between encapsulating underlying tedious technical details to lower programming efforts and exposing indispensable distributed components. The DP enables GreatFree to be adaptable enough to various distributed computing environments.

2) Programming the Simplest Systems

Programming with the DP directly, it is sufficient and necessary to build the most rudimentary distributed system, the Two-Node Client/Server (TNCS) one, which contains two physical distributed nodes and conforms to the interaction principle of the client/server model upon lightweight messaging. Furthermore, it is straightforward to increment the scale of the clients up to the capacity of the server. Then, a more complicated system, which contains multiple clients and a single server, is created. As the counterpart of the TNCS, it is called the Multiple-Node Client/Server (MNCS) system. It proves that the generality of GreatFree because of the axiom that any distributed systems are the aggregation of the TNCS or the MNCS.

3) Programming Distributed Advanced APIs

Based on the generality of GreatFree, besides programming the simplest distributed systems, another primary goal of the DP is used to program the Distributed Advanced APIs (DAA). Although the primitive APIs are generic, it is still expected to create powerful APIs, i.e., the DAA, to raise the programming productivity through further encapsulation. Any DAA is the direct or indirect encapsulation of the DP using the object-oriented technique. Additionally, the procedure is recursive, i.e., any new DAA is built upon programming with the DP or the existing DAA recursively.

As a general-purpose distributed programming paradigm, besides the DP, GreatFree provides additional two categories of DAA, including the distributed clustering and the distributed caching. The distributed clustering is the important programming component to construct scalable distributed systems. The distributed caching provides a large-scale high-performance storage mechanism in the interfaces of common data structures, such as the map, the list, the stack, the queue and so on. Similarly, all of them are derived through programming with the DP and existing DAA recursively.

4) Programming Distributed Frameworks

The motivation to create the DAA aims to program more complicated distributed frameworks rapidly through keeping on hiding low-level details that are unnecessary for one particular distributed context. Once if the DAA is available, it is more efficient to build sophisticated ones.

GreatFree is not an application-level programming paradigm such that it is focused on the establishment of distributed frameworks, which emphasize the semi-constructed distributed systems and ignore the implementation of upper-level applications. Working on a mature framework is currently the primary approach to develop distributed applications. For the complexity of the Internet-oriented computing environment, distributed frameworks have numerous mutations, such as the Peer-to-Peer (P2P), the 3-Tier, the n-Tier, the Map/Reduce, the streaming, the storage, the enterprise cluster, the cloud, and so forth. GreatFree-based frameworks

have one more advantage. Different from the dedicated ones that can hardly be revised, the frameworks of GreatFree can be programmed further conveniently to accommodate to specific requirements on functions as well as performance with the support of the DP and the existing DAA. Over those frameworks, it is easy to establish a great many distributed applications such as chatting, file transmissions, e-commerce systems, gaming, financing, block-chains, and so forth to fulfill various circumstances.

In addition to the common frameworks, GreatFree can be used to program some important distributed frameworks for specific applications, such as the enterprise container, the search engine, the video streaming, distributed file systems, and the distributed data centers. Similarly, those frameworks are programmable further rather than the fixed or configurable ones only. The most complicated distributed systems are the ones dominated by human capital as well as social capital. Such systems emerge with the progresses of the Internet. One example is the World Wide Web (WWW), which is the global scale socialized heterogeneous information system over the Internet. Such a system conforms to the principles of human interactions in addition to those of machines. Thus, the system is highly heterogeneous with potentially infinite users and tremendously high workload. It is impossible to employ any existing techniques to implement it conveniently. Fortunately, because of the natures of the DP, it has already been utilized successfully in the project of the New World Wide Web (N3W). As a highly heterogeneous system, the N3W is one upgraded instance of the global scale socialized distributed system to resolve the drawbacks of the traditional WWW.

B. The ATM

The ATM is a novel concurrency mechanism for distributed programming. There is no way to establish a generic programming paradigm without properly designed threading. To achieve the goal, the ATM is distinct from others in its unique characters, including the visibility, the application-level, the distribution ability, the messaging orientation, and the programmability.

1) The Visible Threading

Threads are the major resource for any distributed programming paradigms since distributed systems are concurrent in nature. It becomes infeasible to conceal or degenerate threads for rapid programming when distributed computing environments become complicated. The characters of the Internet-oriented distributed computing environments result in utilizing resources concurrently in the most fine-grained granularity. Thus, it is required to control threads directly to implement high-quality concurrent algorithms rather than any other management mechanisms. The primary operations on threads include creating, task-assigning, interacting, monitoring, reusing, collecting, and so forth. Only if those functions are available to programmers, it is possible to program sufficiently fine-grained distributed concurrent algorithms to accommodate to the heterogeneity of various distributed circumstances. Following the principle, the ATM provides programmers with the full governance in terms of controlling a single thread in its entire lifecycle.

2) The Application-Level Threading

The application-level threading is defined as a concurrent programming mechanism that provides developers with the independently running threads, which abide by application-level instructions to change their behaviors rather than any system-level commands isolated from upper level scenarios. The application-level threading of the ATM alleviates the difficulty of programming with the system-level threads directly. Through the approach, the ATM threads are programmed via the simplified directives dependent on application progressing statuses rather than taking care of the raw characters of threads. From a programmer's point of view, an instance of the ATM threads is dominated to accomplish various tasks for an application by messages of requesting or eventing until it is overloaded. In accordance with the dynamics of a specific application, programmers are offered the privilege to monitor their current states, evaluate the workload to be assigned to them, and even consider specific scenarios to administrate the thread reasonably. Luckily, all the efforts are focused on the concurrent strategies as

well as the distributed solutions on the upper level instead of the management of the native threading on the lower level. In brief, the ATM is totally different from the system-level approaches that are independent of application scenarios in the SSP.

3) The Distributed Threading

As a large-scale distributed concurrent programming mechanism, the ATM is usually sustained by a scalable distributed cluster made up with multiple slave nodes, which are the ATM thread providers responsible for supplying sufficient ATM threads to fulfill one particular concurrent task. The cluster is accessed by any number of masters, who play the role of an ATM thread consumer. The count of the slaves depends on the computing requirements of specific distributed scenarios such that the scale of the cluster can be enlarged arbitrarily upon workload. To assign concurrent tasks, the master distributes its requirements via asynchronous messaging to the cluster such that the ATM threads originated from the slaves are created, reused, and composed together to accomplish all the subtasks. During the procedure to work on the subtasks, those ATM threads are still able to interact with each other following the commands from the master to deal with additional missions if needed. After one particular task is finished, the final result is gathered from all the ATM threads on the slaves to the master. In brief, rather than a naked thread running asynchronously on a physical standalone computing device in the SSP, an ATM thread is equivalent to a logical distributed node executing scheduled tasks independently in a serial fashion such that it can be exploited with others using various distributed strategies.

4) The Threading on Messaging

The ATM adopts the popular approach, asynchronous messaging, of the MPP to build loosely coupled distributed systems in the heterogeneous distributed computing environments. To guarantee the adaptability, the TVP is another character of the ATM. However, different from other TVP paradigms, with which threads can hardly be manipulated arbitrarily, the ATM is an approach that allows programmers to dominate threads fully on the application level. On the other hand, when programming with the ATM, the visible threading is absolutely not identical to that of the SSP, in which threads are naked for programming such that programmers are required to worry about each detail of threading on the system level. Rather, the ATM is in essence a concurrent mechanism that converts the system-level memory-sharing threading on a physically standalone computer to the application-level message-passing threading over a large-scale distributed computing environment. Programming with the ATM, developers are allowed to create, monitor, reuse and collect the threads from distributed nodes through asynchronous messaging. The messages contain the application-dependent instructions, tasks and states from a thread consumer to thread providers rather than any system-level directives that probably disrupt the upper-level distributed activities.

5) The Programmable Threading

The same as other complicated systems, the ATM is a distributed concurrent programming mechanism that is constructed completely through programming with the DP and the relevant DAA as well. That is another evidence that GreatFree is a generic distributed programming paradigm. In fact, the ATM is an instance of the distributed system implemented with GreatFree. A regular implementation of the ATM is established with a tree-structured cluster, which contains one single collaborator and a lot of children. It is possible that the cluster is overloaded in practice because of heavy tasks. If so, it is convenient to employ an auto-scaling-out cluster to tolerate the potentially high burden on the fly. It is also feasible to update the topology of the cluster for large volume accessing in a wide area. An extreme case is that each node of the cluster is turned from one physical computer to a logical cluster for heavy pressure workload using the DCP of GreatFree. Whatever the implementation is, only GreatFree techniques are sufficient and necessary. As a matter of fact, what can be seen from the perspective of programmers is always a vast number of the ATM threads for them to govern.

C. The DCP

As a discovery in the domain of distributed programming, the DCP reveals that the code structures are steady whatever the heterogeneity is in any specific distributed computing environments. In other words, various heterogeneous distributed systems can be programmed with a limited number of homogeneous code-level design patterns. In particular, no matter how complicated a distributed system, it can be programmed in the homogeneous code structures using the DCP. In GreatFree, any distributed systems, distributed APIs, or distributed frameworks are programmed with the DCP in essence.

1) The Contributions of the DCP

The DCP is a rapid programming approach as it unveils the magic code structures for distributed systems development. It is made up with a limited number of code-level design patterns, which are the steady code structures to compose the DP, the DAA, and distributed frameworks. Each of the patterns plays the role of one particular member of those final systems only. As the DCP represents the fixed code structures, distributed programming with its support is simplified as the procedure to follow the limited number of predefined patterns to assembly distributed APIs. It is no doubt that the solution speeds up distributed systems development.

At first, the DCP discloses that GreatFree is a rapid distributed programming paradigm that provides sufficient and necessary building block. For the homogeneity of the DCP, the programming effort is lowed obviously. In other words, GreatFree is the craftily simplified solution to sustain the balance between the ease of distributed programming with the DCP and the coverage of distributed computing environments. GreatFree does not intend to conceal all the distributed techniques because of the complexity of the Internet-oriented computing environments.

In addition, the DCP reveals that a generic solution is achievable since the variety of heterogeneous distributed systems adhere to the common principle that they are homogeneous in terms of the distributed code structures. If the principle is luckily founded, the solution is certainly invented. GreatFree is no doubt such a generic programming paradigm. More important, the DCP demonstrates that it is practical to propose a generic and rapid distributed programming language which relies on GreatFree. For the sake of popularity, one choice is the object-oriented script although it is not the unique choice. Although it is impossible to conceal all the technical details of the Internet-oriented computing environment, it is feasible to abstract them in the same forms with the common code structures.

2) The Internal and External Patterns

Using GreatFree, after one distributed algorithm for one particular domain is investigated clearly, the approach to specify it is straightforward since the unique task left is to assemble the primitive distributed programming components. The procedure is equivalent to the one to construct a new DAA or a distributed framework, i.e., programming distributed algorithms with GreatFree results in high-level models. In other words, either a new DAA or a new distributed framework is created through aggregating the DP as well as existing DAAs upon the DCP with respect to the corresponding distributed algorithms. During the procedure, the DCP is the unique series of components to aggregate various distributed resources and mechanisms. In brief, the internals of any DAA and distributed frameworks in GreatFree are implemented through programming with the DCP.

In contrast, any newly created DAA has its own code-level design pattern, i.e., the idiom that encloses the API for rapid programming. Compared with those internal ones to form the new DAA, the pattern is called the external one since it is employed for the implementation that weaves itself outside with other distributed programming components to construct more complicated ones. Each DAA is similar as each DAA is implemented by low-level components in the DCP. Therefore, each DAA either keeps one of the DCPs as its external pattern or reconstructs a new pattern which is a straightforward aggregation of some of the DCP. No any new code-level design patterns are invented for any new DAAs no

matter how complicated a DAA is. That proves that external patterns for DAAs conform to the principle of DCP as well. In other words, the DCP is a self-similar system. For a newly created distributed framework, it does not make sense to discuss about its patterns since it is a semi-constructed system in which no additional distributed programming efforts left except specifying applications and reusing the DCP before a distributed system is established. However, it is feasible to extract the core of one distributed framework to create a new DAA.

IV. THE CLOUD-SIDE OPERATING SYSTEM

The cloud-side operating system is inspired by GreatFree. Since GreatFree is a generic distributed programming language which represents the common principles of any distributed systems over the Internet, it indicates that it is feasible to build a new cloud system that is a generic development and running environment for any distributed systems. On the other hand, as a new operating system, similar to traditional ones, it is necessary to have a proper language to support applications developments on it. Without doubt, GreatFree exhibits the proper choice to be competent to play the role.

A. The Relationships Between Programming Languages and Operating Systems

The relationships between programming languages and operating systems are concluded as follows. At first, a operating system needs to be implemented with a programming language. Additionally, after the operating system is constructed, the same programming language is required to be the technique to develop upper-level applications on it. In other words, without an appropriate programming language, any operating system can hardly be established, and the operating system is useless since it is only a development and running environment without any applications which end users can access.

A programming language is a series of common representations to describe and manage computing resources in one particular computing environment. It is highly recommended that the representations are written in the format that is as human-readable as possible such that developers can program with them conveniently. An operating system is a development and running environment that fits the computing circumstance exactly. Therefore, the system can be implemented rapidly with the language only. Any other low-level languages must bring heavy workloads for sure and any other high-level ones can never support the establishment of such a system.

Once if the operating system is constructed, it speeds up applications development in the same environment. Usually, many semi-constructed frameworks created by the language are preinstalled on the operating system such that they lower the efforts of application programmers extraordinarily. In most cases, developers focus on application level specifications only when working with those tools. However, it is possible that those tools cannot provide some complicated developments with sufficient supports. Then, the programming language is the last choice to overcome the potential barriers in those cases. Although the development efforts are higher than using those frameworks, it is still a feasible solution compared with those languages that are not focused specially on the particular computing environment. In practice, if the difficult cases are used frequently, new frameworks are created upon the programming language such that other programmers enjoy the convenience of the new frameworks.

The combination of UNIX/C is the most well-known example to present the relationships between a programming language and the operating system. Initially, C is a system programming language to specify algorithms that fit the standalone and sequential computing environment. Most code of UNIX is written in C, and it is tough to implement such a complicated system with earlier generation languages, such as assembly ones. After UNIX is constructed, it is a common sense that many function libraries are available over the platform for particular applications developments. Furthermore, during the procedure of UNIX's popularization, a huge bunches of function libraries were implemented with C to ease applications developments over UNIX.

B. The Problems of Traditional Programming Languages and Operating Systems

With the development of Internet technology, most applications are required to run in the concurrent and distributed manner rather than the standalone and sequential one. Unfortunately, because no proper programming languages were available in the past days, the standalone and sequential languages played the major role to program various distributed systems with the support of networking and threading. The procedure is notoriously difficult because developers are forced to make every effort to convert the standalone and sequential programs to the distributed and concurrent ones.

Even though many distributed frameworks are created to lower the workload of distributed systems development, there is no way to modify them to adapt to new environments conveniently. To build a distributed system with low costs, instead of programming with a single language, a couple of third-party heterogeneous frameworks are put together roughly with inefficient protocols, such as HTTP/JSON, without knowing internals of each of them. If the system to be implemented is a large scale one, a lot of heterogeneous frameworks have to be pieced together. That is the nightmare of developers. In fact, because of the native drawbacks of traditional languages, it is a tough job for each developer to implement the simplest distributed system. Although frameworks help, because of the complexity of the Internet-oriented computing environment, it is impossible to program any distributed systems from scratch in most cases. However, piecing heterogeneous frameworks together always results in poor adaptability, low performance, high costs and maintenance difficulties.

The above problems also unveil that the current operating systems are not the proper development and running environment for distributed systems over the Internet-oriented computing environment. Since those operating systems are implemented with standalone and sequential languages, they do not provide distributed and concurrent systems with sufficient supports. That is the primary reason that almost each distributed application needs to be developed and run over frameworks rather than those operating systems directly. Because of that, those heterogeneous distributed frameworks are called the middleware layer between applications and the operating systems. The larger the scale of a distributed system, the more complicated the middle layer. It is not difficult to imagine the heavy overhead of computing resources consumption and the chaotic architectures.

In brief, at present, the fundamental software in terms of operating systems as well as programming languages is not well established for distributed systems' development and running.

C. The Concept of Cloud-Side Operating System and GreatFree

The cloud-side operating system is a generic development and running environment for distributed systems over the Internet-oriented computing circumstance. At this moment, such a system is still not available. When talking about the term of operating systems, it always represents the traditional ones, such as UNIX and Windows, which are viewed as the development and running platforms for standalone and sequential applications. Because of their native drawbacks, they are improper choices to support distributed systems development and running.

As a counterpart of traditional ones, the idea of the cloud-side operating system is originated from the generic programming language, GreatFree. At present, GreatFree is becoming more and more mature in the domain of distributed programming over the Internet. For that, it inspires the establishment of the cloud-side operating system. With its distinct characters for distributed programming, GreatFree is not only the correct technique to implement the cloud-side operating system but also the right choice to program upper level applications over the same platform.

V. CONCLUSIONS

By now, we have completed a complete delivery room procedure. This process is the core work of implementing the server using the distributed elements of GreatFree. You can see that all the programs involved are written according to the design patterns provided by GreatFree. As a beginner, there must be a process

of adaptation to these patterns. But at least the process is straightforward. With traditional languages, accomplishing this task is uncertain and unwieldy, and even the most sophisticated programmers don't want to tread lightly. Lonely Chatter is just the simplest distributed system, but it gets harder in more complex distributed scenarios. In contrast, GreatFree has provided a set of design methods with consistent ideas, clear steps and stable forms. More importantly, it can be used for any distributed problem. No matter what system, these patterns are used repeatedly from simple to complex. This reflects the unique point of GreatFree technology.

REFERENCE

[1] J. V. Guttag, "Introduction to Computation and Programming Using Python", the MIT Press, ISBN: 978-0-262-52500-8, 2013.

[2] A. Gupta, "Java EE 7 Essentials", O'Reilly Media, ISBN: 978-1-449-37017-6, 2013.

[3] A. Goncalves, "Beginning Java EE 7", Apress, ISBN-10: 143024626X, ISBN-13: 978-1430246268, 2013.

[4] S. Newman, "Building Microservices - Designing Fine-Grained Systems", O'Reilly, ISBN: 978-1491950357

[5] C. Richardson, "Microservice Patterns", Manning Publications, ISBN-10: 1617294543, ISBN-13: 978-1617294549, 2018.

[6] Apache Whisk, https://openwhisk.apache.org

[7] AWS Lambda, https://aws.amazon.com/lambda

[8] IBM Cloud Functions, https://www.ibm.com/cloud/functions

[9] Google Cloud Functions, https://cloud.google.com/functions

[10] Microsoft Azure Functions, https://azure.microsoft.com/services/functions

[11] Oracle Fn Functions, https://fnproject.io

[12] Service-Oriented Architecture Standards – The Open Group, https://www.opengroup.org/forum/service-oriented-architecture-soa

[13] M. Bell, "Introduction to Service-Oriented Modeling, Service-Oriented Modeling: Service Analysis, Design, and Architecture", Wiley & Sons, ISBN: 978-0-470-14111-3

[14] T. White, "Hadoop: The Definite Guide", the Third Edition, O'Reilly, ISBN: 978-1-449-32891-7, 2012.

[15] S. Ghemawat, H. Gobioff, S. T. Leung, "The Google File System", Proceedings of the 19th ACM SOSP, Pages: 29-43, 2003.

[16] V. Jason, "Pro Hadoop", Apress, ISBN: 978-1-4302-1942-2, 2009.

[17] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, I. Stoica, "Spark: Cluster Computing with Working Sets", Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, 2010.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstractions for In-Memory Cluster Computing", Technical Report UCB/EECS-2011-82, EECS Department, University of California, Berkeley, 2011.

[19] M. Luksa, "Kubernetes in Action", Manning, ISBN-13: 978-1617293726, ISBN-10: 9781617293726, 2018.

[20] J. D. Moore, "Kubernetes: The Complete Guide To Master Kubernetes", Independently Published, ISBN-10: 1096165775, ISBN-13: 978-1096165774, 2019 (not downloaded yet. 05/20/2019, LB).

[21] A. Shrivastwa, S. Sarat, K. Jackson, C. Bunch, E. Sigler, T. Campbell, "OpenStack: Building a Cloud Environment", Packt Publishing, ISBN-10: 1787123189, ISBN-13: 978-1787123182, 2016.

[22] B. Silverman, M. Solberg, "OpenStack for Architectures: Design Production-Ready Private Could Infrastructure", the Second Edition, Packt Publishing, ISBN-10: 1788624513, ISBN-13: 978-1788624510, 2018.

[23] D. R. Butenhof, "Programming with POSIX Threads", Addison-Wesley, ISBN: 0-201-63392-2, 1997.

[24] B. Nichols, D. Buttlar, J. Farrell, "Pthreads Programming", O'Reilly, ISBN: 1-5692-115-1, 1996.

[25] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, D. Lea, "Java Concurrency In Practice", Addison-Wesley Professional, ISBN-10: 0-321-34960-1, ISBN-13: 978-0-321-34960-6, 2006.

[26] D. Lea, "Concurrent Programming in Java, Design Principles and Patterns", the Second Edition, Addison-Wesley, ISBN: 0-201-31009-0, 1999.

[27] S. Cleary, "Concurrency in C# Cookbook, Asynchronous, Parallel, and Multithreaded Programming", O'Reilly, ISBN: 978-1-449-36756-5, 2014.

[28] C. Hughes, T. Hughes, "Parallel and Distributed Programming Using C++", Addison-Wesley, ISBN: 0-13-101376-9, 2003.

[29] R. Terrell, "Concurrency in .NET, Modern Patterns of Concurrent and Parallel Programming", Manning, ISBN: 978-1-617-29299-6, 2018.

[30] H. Okamura, M. Tokoro, "The Design and Implementation of ConcurrentSmalltalk", Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications, Pages: 331-340, 1986.

[31] Y. Yasuhiko, "The Design and Implementation of ConcurrentSmalltalk", Proceedings of Conferences on Object-Oriented Programming Systems, Languages and Applications, Pages: 331-340, 1986.

[32] H. Okamura, M. Tokoro, "ConcurrentSmalltalk-90", Proceedings of TOOLS Pacific'90, 1990.

[33] I. Balbaert, "Rust Essentials", Packt Publishing, ISBN: 978-1-78528-576-9, 2015

[34] G. Zaccone, "Python Parallel Programming Cookbook", Packt Publishing, ISBN: 978-1-78528-958-3, 2015.

[35] A. Shrivastwa, S. Sarat, K. Jackson, C. Bunch, E. Sigler, T. Campbell, "OpenStack: Building a Cloud Environment", Packt Publishing, ISBN-10: 1787123189, ISBN-13: 978-1787123182, 2016.

[36] B. Silverman, M. Solberg, "OpenStack for Architectures: Design Production-Ready Private Could Infrastructure", the Second Edition, Packt Publishing, ISBN-10: 1788624513, ISBN-13: 978-1788624510, 2018.

[37] K. Jackson, C. Bunch, E. Sigler, J. Denton, "OpenStack Cloud Computing Cookbook", Packt Publishing, ISBN-10: 1788398769, ISBN-13: 978-1788398763, 2018.

[38] J. Rutherglen, D. Wampler, E. Capriolo, "Programming Hive", O'Reilly, ISBN: 978-1-449-31933-5, 2012.

[39] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, M. Zaharia, "Spark SQL: Relational Data Processing in Spark", Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Pages: 1383-1394, 2015.

[40] A. Sarkar, "Learning Spark SQL: Architect Streaming Analytics and Machine Learning Solution", Packt Publishing, ISBN-10: 1785888358, ISBN-13: 978-1785888359, 2017.

[41] F. Chang, et. al., "Bigtable: A Distributed Storage System for Structured Data", Journal of ACM Transaction on Computer Systems (TOCS), Volume 26, Issue 2, Article No. 4, Pages: 4:2-4:26, 2008.

[42] N. Dimiduk, A. Khurana, "HBase In Action", Manning Publications, ISBN: 978-1617290527, 2012.

[43] L. Georgo, "HBase: The Definitive Guide", O'Reilly Media, ISBN: 978-1-449-39610-7, 2011.

[44] S. Akhtar, R. Magham, "Pro Apache Phoenix: An SQL Driver for HBase", the First Edition, Apress, ISBN-10: 9781484223697, ISBN-13: 978-1484223697, 2016.

[45] M. Kornacker, et. al., "Impala: A Modern, Open-Source SQL Engine for Hadoop", Proceedings of the 7th Biennial Conference on Innovative Data Systems Research (CIDR'15), 2015.

[46] J. Russell, "Getting Started with Impala", ISBN-10: 1491905778, ISBN-13: 978-1491905777, O'Reilly Media, 2015.

[47] A. Katsifodimos, S. Schelter, "Apache Flink: Stream Analytics at Scale", Proceedings of 2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW), Pages: 193-193.

[48] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, "Apache Flink: Stream and Batch Processing in a Single Engine", Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, Volume 36, No. 4, Pages: 17-29, 2015.

[49] F. Hueske, V. Kalavri, "Stream Processing with Apache Flink", O'Reilly Media, ISBN-10: 149197429X, ISBN-13: 978-1491974292, 2019.

[50] K. M. M. Thein, "Apache Kafka: Next Generation Distributed Messaging System", International Journal of Scientific Engineering and Technology Research, ISSN: 2319-8885, Volume: 03, Issue: 47, Pages: 9478-9483, 2014.

[51] N. Garg, "Apache Kafka", Packt Publishing, ISBN: 978-1-78216-793-8, 2013

[52] S. T. Allen, M. Jankowskl, P. Pathirana, "Storm Applied: Strategies for Real-Time Event Processing", Manning Publications, ISBN-10: 1617291897, ISBN-13: 978-1617291890, 2015.

[53] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, P. Poulosky, "Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming", Proceedings of 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Pages: 1789-1792 (not downloaded yet, 06/24/2019, LB).

[54] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, "Pig-Latin: A Not-So-Foreign Language for Data Processing", Proceedings of ACM SIGMOD International Conference on Management of Data, Pages: 1099-1110, 2008.

[55] A. Gates, D. Dal, "Programming Pig: Dataflow Scripting with Hadoop", O'Reilly Media, ISBN-10: 9781491937099, ISBN-13: 978-14919337099, 2016.

[56] M. Islam, A. K. Huang, M. Battisha, M. Chiang, S. Srinivasan, C. Peters, A. Neumann, A. Abdeinur, "Oozie: Towards a Scalable Workflow Management System for Hadoop", Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies, Pages: 4-13, 2012.

[57] M. K. Islam, A. Srinivasan, "Apache Oozie: The Workflow Scheduler for Hadoop", ISBN-10: 1449369928, ISBN-13: 978-1449369927, 2015.

[58] D. Smiley, E. Pugh, K. Parisa, M. Mitchell, "Apache Solr Enterprise Search Server", the 3rd Edition, Packt Publishing, ISBN: 978-1-78216-136-3, 2015.

[59] A. Serafini, "Apache Solr: Beginner's Guide", Packt Publishing, ISBN: 978-1-78216-252-0, 2013.

[60] J. Brittain, I. F. Darwin, "Tomcat: The Definitive Guide", the 2nd Edition, O'Reilly Media, ISBN-10: 0-596-10106-6, ISBN-13: 978-0596-10106-0, 2007.

[61] D. Thomas, "Programming Elixir >= 1.6: Functional |> Concurrent |> Pragmatic |> Fun", Pragmatic Bookshelf, ISBN-10: 1680502999, ISBN-13: 978-1680502992, 2018.

[62] S. Juri, "Elixir In Action", the 2nd Edition, Manning Publications, ISBN-10: 1617295027, ISBN-13: 978-1617295027, 2019.

[63] J. Armstrong, "A History of Erlang", Proceedings of the Third ACM SIGPLAN Conferences on History of Programming Languages, Pages: 6-1 – 6-26, 2007.

[64] J. Armstrong, "The Development of Erlang", Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming, Pages: 196-203, 1997.

[65] J. Armstrong, "Making Reliable Distributed Systems in the Presence of Software Errors", PhD Dissertation, Royal Institute of Technology, 2003.

[66] J. Armstrong, "Erlang", Communications of the ACM, Volume: 53, No. 9, Pages: 68-75, 2010

[67] J. Armstrong, R. Virding, C. Wikstrom, M. Williams, "Concurrent Programming in Erlang", the 2nd Edition, Prentice Hall, ISBN-10: 013508301X, ISBN-13: 978-0135083017, 1996.

[68] J. Armstrong, "Programming Erlang: Software for a Concurrent World", the 2nd Edition, Pragmatic Bookshelf, ISBN-13: 978-1-937785-53-6, 2013.

[69] F. Cesarini, S. Thompson, "Erlang Programming: A Concurrent Approach to Software Development", O'Reilly Media, ISBN-10: 0596518188, ISBN-13: 978-0596518189, 2009.

[70] V. A. Sarawart, K. Kahn, J. Levy, "Janus: A Step Towards Distributed Constraint Programming", Proceedings of the 1990 North American Conference on Logic Programming, Pages: 431-446, 1990.

[71] V. A. Saraswat, M. Rinard, P. Panangaden, "The Semantic Foundations of Concurrent Constraint Programming", Proceedings of Ninth ACM Symposium on Principles of Programming Languages, Pages: 333-352, 1991.

[72] D. Gudeman, S. K. Debray, K. DeBosschere, "jc: an Efficient and Portable Sequential Implementation of Janus", Proceedings of the International Conference and Symposium on Logic Programming, Pages: 399-416, 1992.

[73] Red Programming Language, https://www.red-lang.org

[74] C. Varela, G. Agha, “Programming Dynamically Reconfigurable Open Systems with SALSA”, Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Pages: 20-34, 2001

[75] B. Nobakht, F. S. de Boer, “Programming with Actors in Java 8”, Proceedings of Leveraging Applications of Formal Methods, Verification and Validation, Specialized Techniques and Applications, Pages: 37-53, 2014

[76] Akka, https://akka.io

[77] M. K. Gupta, “Akka Essentials”, Packt Publishing, ISBN-10: 1849518289, ISBN-13: 978-1849518284, 2012

[78] D. Wyatt, “Akka Concurrency”, Artima Inc., ISBN-10: 0981531660, ISBN-13: 978-0981531663, 2012

[79] R. Roestenburg, R. Bakker, R. Williams, “Akka in Action”, Manning Publications, ISBN-10: 1617291013, ISBN-13: 978-1617291012, 2016

[80] V. Vernon, “Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka”, Addison-Wesley Professional, ISBN-10: 0133846830, ISBN-13: 978-0133846836, 2015

[81] P. Haller, F. Sommers, “Actors in Scala”, Artima Inc., ISBN-10: 0981531652, ISBN-13: 978-0981531656, 2012

[82] N. Raychaudhuri, C. Fowler, “Scala in Action”, Manning Publications, ISBN-10: 1935182757, ISBN-13: 978-1935182757, 2013