GREATFREE: THE JAVA APIS AND IDIOMS TO PROGRAM LARGE-SCALE DISTRIBUTED SYSTEMS

Bing Li

Department of Computer Science and Engineering, Xi'An Technological University, Xi'An, China

ABSTRACT

This paper introduces a series of APIs and idioms in Java SE (Java Standard Edition), GreatFree, to program large-scale distributed systems from scratch without adopting any third party frameworks. When programming with GreatFree, developers are required to take care of rather than be invisible to most of the implementation issues in a distributed system. It not only strengthens developers' skills to polish a system but also provides them with the techniques to create brand new and creative systems. However, taking care of many such issues is a heavy load because of the low-level of Java SE. To alleviate the burden to program with Java SE directly, GreatFree provides numerous APIs and idioms in Java SE to help programmers resolve indispensable distributed problems, such as communication programming, serialization, asynchronous and synchronous programming, resource management, load balancing, caching, eventing, requesting/responding, multicasting, and so forth. Additionally, as an open source tool to program, developers are able to strengthen their systems through not only adjusting GreatFree parameters but also upgrading GreatFree APIs and idioms themselves. According to the current intensive experiments, it is convenient for developers to program an ordinary or a large-scale distributed system from scratch with GreatFree.

KEYWORDS

Design Patterns, Distributed Systems, Wireless Network, Idioms, Application Program Interface, Distributed Programming, Concurrency

1. INTRODUCTION

The paper exhibits an open-source programming tool, GreatFree, to assist developers to program large-scale distributed systems from scratch. Based on Java SE, GreatFree includes a series of APIs and idioms to ease the programming procedure. Nowadays most developers avoid programming a distributed system directly because of many issues to be resolved. In contrast, it is more convenient to configure mature open sources and commercial frameworks than to program with generic low-level programming languages like Java SE. However, in many specific cases it needs to implement a system by programming rather than configuring. Professional developers are required to be competent with dealing with complicated problems in a distributed computing environment through programming.

To implement a distributed system by programming, it encounters many implementation barriers, such as communication [1], serialization [1], asynchronous and synchronous management [2][3], resource management [4], load balancing [5], caching [6], eventing and requesting/responding [7], multicasting [8], and so forth. It is tough to implement such a system with generic fundamental programming languages like Java SE. Nevertheless, programming a distributed system from scratch is required in many specific cases. First of all, a brand new creative system requires developers to program the low-level models themselves to construct a self-contained

DOI : 10.5121/ijait.2016.6101

system other than to resolve high-level problems only. The goal can never be achieved with the approaches of configuring and customizing. Second, a system needs to be flexible enough to be customized for a specific issue by programming. Though it is seldom that all of the distributed features be implemented from scratch, a limited change is frequently required in a system. It is convenient for programmers to handle that only if all of the code is open and easy to be changed. Third, for professional developers, it is required for them to be competent with the task of programming a complicated system without the assistance of mature frameworks. Fourth, for security reasons, some systems, such as a creative one in a new environment, are forced to implement from an initial level. For those considerations, a bunch of APIs and idioms, which is named GreatFree, are proposed to assist developers to program distributed systems, specially the large-scale ones.

The methodology GreatFree represents encourages proficient developers to program their own distributed systems rapidly rather than to configure a mature framework. It provides developers with a new programming tool that focuses on the large-scale distributed computing environment. To achieve the goal, it prevents developers from heavy loads of programming a complicated system with fundamental programming languages, such as Java SE (Java Standard Edition) [9]. For that, it comes up with a bunch of APIs (Application Program Interfaces) and design patterns as idioms to solve the specific distributed issues such as client/server [10] and peer-to-peer models [10], distributing eventing [7] and polling [7], resource management [4], distributed concurrency and synchronization [2][3], distributed multicasting [8], distributed clusters [10] for computing and memorizing, and so forth. With the support of the GreatFree APIs and idioms, it is convenient for developers to program a large-scale distributed system from scratch. In brief, GreatFree is a programming environment that resides between the mature configurable systems and the naked generic programming languages. Using it, a high quality distributed system can be conveniently programmed from scratch.

When using GreatFree, it is required for developers to keep most of the distributed issues in mind. That is different from the traditional opinions of software engineering, which emphasizes the goal to hide all of the details of low-level systems such that developers are invisible to the specific computing environments. They need only to change limited parameters or configurations following tutorials and guiding books without worrying about what happens underlying high-level applications. On the contrary, the methodology of GreatFree proposes that most important techniques should be discernible to developers such that they need to resolve them by programming with relevant APIs and patterns. Developers are required to compose those stuffs to form an arbitrary application. Although mature systems minimize developers' effort, they also lower developers' abilities such that they can hardly deal with changes if they do not program for a long-term. Such developers without sufficient knowledge might be able to customize a high quality system by configuring mature systems. But they will never be able to implement a creative system in a new domain. Therefore, using GreatFree, developers always need to learn and keep in mind the relevant technologies. And then, when they would like to implement a brand new system, the methodology of GreatFree become their potential silver bullet.

On the other hand, GreatFree is different from the naked generic programming languages like Java SE. First of all, it contains a series of domain-specific APIs and idioms, which focus on the area of distributed systems, especially the large-scale ones whereas a generic programming language is required to deal with all of the issues in any domains. The convenience of GreatFree is that developers' programming loads are reduced magnificently for its well-defined APIs and idioms. Although they need to be aware of the issues of distributed systems, they are not required to implement them using Java SE. On the contrary, they just need to program in a composing manner with the APIs and idioms from GreatFree. More important, proficient programmers are encouraged to revise the code of the APIs and idioms because all of the source code of GreatFree is open [11].

For the distinguished character of GreatFree, developers of the programming tool need to learn the distributed issues at first. To be invisible to the problems, they must have no idea which APIs and idioms should be chosen to program with a composing style. Fortunately, the programming procedure looks like constructing a tower with high quality building blocks rather than sands and stones directly.

The main contributions of the paper are summarized as follows.

As a methodology, a developer needs to learn the core techniques of the system they are programming rather than to configure a mature system without knowing anything about its internal principles. This is especially important to design a brand new and creative system.
To lower the burden of developers' effort, open source based APIs and idioms are effective in terms of its high quality and high changeability.

According to the above opinions, a series of APIs and idioms are proposed to help developers program a large-scale distributed system.

The sections of the paper are organized as follows. Section 2 gives an introduction to the related work, including configuring mature frameworks, design patterns and programming scratch. Section 3 addresses the primary methodologies of GreatFree. Section 4 presents GreatFree APIs to support distributed system programming. Section 5 discusses GreatFree idioms that compose GreatFree APIs. Section 6 explains the experiment environment of GreatFree APIs and idioms. Section 7 describes the future work.

2. RELATED WORK

In the domain of software engineering, to lower the burden to implement a distributed system, many mature frameworks are put forward as open sources [12][13][14][15] or commercial products [16] for specific application environments. Thus, developers are not required to program but configure those systems to fulfill their requirements. In addition, although some generic programming languages [9] are strengthened gradually to lower the cost to program, their distances to a distributed system are still too long to be reached by coding directly for most programmers. Finally, design patterns [17][18][19][20][21][22] are also believed to be one approach to implement a complicated system like a distributed one. However, until now, most design patterns are not the code in specific languages to be reused conveniently. Although idioms, as small-scale design patterns, intend to resolve the issue, a comprehensive solution of idioms is not available in the domain of distributed systems.

2.1. Configuring Mature Systems

Nowadays when implementing a distributed system, a common phenomenon is that developers get accustomed to downloading existing mature open sources or commercial systems rather than programming themselves line by line. No matter whether a system is complicated enough like a so-called cloud [15] or even a fundamental one like a client/server model [10], most developers give up implementing a system through programming. Although each of them must take the relevant classes or trainings in universities or other schools, they become scared about programming those systems themselves.

The primary reason is due to the fact that those mature systems could fulfill their requirements. For example, to set up a Web site, Tomcat [12] is one of the popular choices as a Web server. Even for a gaming system, a Web server like Tomcat is suitable to support its centralized

information exchanging among users. To initiate an e-commerce online system, developers prefer the Java EE (Java Enterprise Edition) [13] to the generic programming language, Java SE (Java Standard Edition) [9], since the former one hides most details of distributed techniques for enterprises from developers.

In addition, it is difficult to program high quality distributed systems by programming from scratch. Developers have to be aware of numerous technical details, such as network communication [1], serialization [1], asynchronous and synchronous programming [2][3], caching [6], eventing and requesting/responding [7], resource management [4], load balancing [5], multicasting [8], and so forth. All of the issues turn out to be a heavy workload to program.
Finally, most mature systems are adaptable to subtle changes in their specific environments. Those systems allow developers to configure in the event that some requirements cannot be fulfilled by default. They provide developers with configuration files in an XML [23] format such that they can set up their own preferences through modifying relevant parameters. For example, to update the Solr [24] configuration file, developers are able to construct a tree-like structure to distribute searching loads. When using JBoss [25], WebSphere [26] or other application servers, business transactions [6] are required to be specified in configuration files.
In short, when the requirements are clear and the relevant mature frameworks are available to the specific environment, it is reasonable to select an appropriate commercial framework or download an open source system to accomplish the goal rapidly.

2.2. Disadvantages of Configuring Mature Systems

However, besides the advantages of mature frameworks, developers should be aware of the disadvantages when using them to implement their systems. First of all, all of those mature systems are specific to their respective particular domains and their adaptability is restricted to narrow scopes. Although most existing frameworks claim they are adjustable to different environments, it is always an impossible mission when using the systems to a domain out of their ranges. For example, no one ever believes Java EE based enterprise systems could be used to take the load of sharing online videos. If a conventional Web server, like Tomcat [12], could work efficiently to transmit high-volume data, why some browser plug-in systems, like FlashGet [27], exist? Although online Web chatting systems were popular in 1990s, they were completely replaced by instant messaging [28]. Even though instant messaging systems dominate the market of lightweight information exchanging, they are almost beat by socialized systems, like Twitter [29], eventually. Therefore, once if a framework is designed, its adaptation is usually constrained tightly within its domain. In brief, mature frameworks can hardly adapt to the updates in their preferred computing environment.

Second, even though within the preferred specific domain for their designs, their adaptability is limited in terms of the obvious differences between configuring and programming. For example, in Solr, using its configuration files, developers are offered the privilege to customize a hierarchical distributed structure to achieve the goal of load balancing. Indexed data is transmitted among the nodes by the protocol of HTTP [30]. Each node in the structure has to pull its parent node periodically to obtain latest data. However, the drawback is apparent in the case. First, developers are not free enough to construct an arbitrary topology other than the hierarchical one. Moreover, only the protocol of HTTP is used between the nodes. The periodically pulling is not efficient as an eventing and streaming based protocol [10], especially when the timing issue is critical in a particular application. Because of the limitations of the Solr, developers can do nothing. Only programming can achieve the goal to upgrade underlying protocols and algorithms. Mature systems are not adaptable enough to fulfill the requirements. Compared with programming, the flexibility upon configuring is believed to be extremely limited.

Third, developers probably rely on those systems such that it is difficult for them to keep and improve their skills. Using them, developers need only to describe requirements and change configurations. It can be achieved with few programming experiences. For example, to implement an e-commerce system with a Java EE application server [13][26], only business logic is required for developers to write code in the object-oriented model [17]. It is unnecessary for them to worry about threading [2][3], data transmission [1], synchronizing [6], resource management [4], state management [6], transaction management [6] and so forth. From the perspective of software engineering, it lowers developers' cost and speed up the system development. However, those developers always lose the indispensable expertise during the procedure. For some beginners, they even ignore the required knowledge and skills. After a period, it is impossible for them to be competent with implementing a creative system from scratch.

Fourth, sometimes those mature systems are easily misused and excessively utilized in inappropriate environments. The problem often happens when developers deploy those systems to unsuitable domains. Even though they become aware of the potential problems, the convenience of deploying and configuring attracts them to avoid the troubles of programming. Sometimes such a misusing might work, but that is usually a high cost and cumbersome solution. For example, to set up a peer-to-peer (P2P) [10] model in a distributed environment, developers would rather install Tomcat [12] on each node as the server to receive data and then find a HTTP client [30] for each of them to send data. The solution works since data can be sent from the client to the server by the HTTP request and any nodes are able to play the role of a server to receive remote requests and notifications. However, Tomcat is a huge stuff itself. It contains many other irrelevant components that are never used in the P2P system, such as JavaServer Pages [31]. Moreover, Tomcat consumes a large portion of resources as a Web based application platform on a server that has rich computing resources. Thus, it influences the quality of the P2P system, especially when it is deployed to the devices whose resources are limited. In most time, a P2P system needs to deal with a large number of heterogeneous computing devices. Most of them lack computing resources. Therefore, Tomcat is unsuitable to the case. To some extent, the above solution is not bad although it is far from perfect. The most typical mistake is that developers would rather use Web servers anywhere for communications. The problem is often seen in the case that developers lack rich experiences, especially who scares about the TCP programming and the concurrency programming.

Finally, there are always new domains and novel ideas that are not covered by those mature systems. A mature system emerges only when the specific application environment becomes dominant and its primary requirements keep steady for a long period. For example, the so-called application server is usually suitable to the centralized mission-critical enterprise distributed environment to support e-commerce. Hadoop is proposed for the high concurrency and large-scale distributed clusters that primarily deal with high volume read-only data. Fortunately, with the progress of the computing world, new domains or computing environments always come out. The mobile Internet is apparently such an instance. Many fancy applications are to be implemented to accommodate the specific context. For example, when developers are designing an application to present Web pages on a smart phone, they have to take into account to accomplish the task with programming rather than simply to embed a browser into their applications. Thus, many Web sites nowadays design their own clients on smart phones and encourage users to access their sites using the specially customized clients rather than using traditional generic browsers. Because of the specific designs, the clients provide users with high quality accessing experiences. In addition, when new ideas are available about a specific topic, traditional approaches must be out of date. It is required for developers to keep programming to implement additional modules and integrate them with others. For instance, when a new routing algorithm [32] is available for a P2P system [33], it is impossible to exploit it by configuring existing systems. Developers have to implement the algorithm by programming.

In brief, although mature software systems are convenient for developers to deal with requirements in a particular computing environment, for a software developer, it is required to keep in mind that programming instead of configuring is the only way to raise their ability to deal with all kinds of difficult problems. In some specific scenarios, programming line-by-line is mandatory to the success of a system, especially when it is a brand new one in terms of scientific or business creations.

2.3. Design Patterns

For the problem of programming languages, some senior researchers and engineers propose design patterns [17][18][19][20][21][22] to assist junior programmers. Design patterns aim to provide developers with a bunch of mature solutions written in pattern languages [21][22] to resolve recurring problems in various domains, such as the object-oriented (OO) programming context [17], the distributed computing [18] and the enterprise systems [18]. The patterns are proved to be effective. For example, to the issues of OO programming, patterns like creational, structural and behavioral [17] ones lower developers' effort to reinvent them themselves through a long-term practice. Unfortunately, those stuffs retain to be generic in all of the above domains. Thus, design patterns are not regarded as the final solution that can be reused rapidly to solve the problems of specific computing environments. Instead, they need to spend high effort to be aware the contexts where those abstract patterns are suitable. After that, those patterns have to be transformed into specific code. During the process, programmers are never isolated from the implementation details. It is still a tough job. In the domain of distributed systems, the burden becomes severely heavier.

For a particular type of patterns, idioms are defined as the ones written in specific languages. If so, the effort to program with them must be low. However, a comprehensive solution to the topic of distributed systems is not available to the best of my knowledge. In the Java world, such a solution is also almost unavailable. Even though some exists, it is still fine for developers to taste the solution of GreatFree and make a better choice.

2.4. Programming from Scratch

Most developers who lack rich experiences are scared about programming directly, especially when implementing a distributed system from scratch. Programming from scratch is designated as the developing approach through which developers take into account most of the specific issues of a particular application domain and solve them by coding themselves rather than importing existing modules from third parties. It seems that the approach violates the convention of software engineering that encourages developers to reuse mature legacy code and systems. In fact, that is not the truth all the time.

In the following three cases, programming is still required. First, reusable legacy resources are unavailable in a specific domain. Second, changes are required to fit a particular situation. Third, new ideas or new environments emerge such that it is required to come up with a relatively independent system. To deal with the issues of the above cases, programming becomes the indispensible skills for developers.

The primary reason that developers dislike programming from scratch is that most languages, such as Java SE, are generic to various application domains such that it is short of effective modules, fully grown APIs and compelling patterns to deal with specific issues. Even though developers grasp all of the details of a language and learn the requirements and solutions of the domain, it still takes them a high cost to program a high quality system in a specific area directly.

3. THE INTENTIONS OF GREATFREE

GreatFree is made up with a series of APIs and idioms to overcome the difficulties of programming large-scale distributed systems. It believes the reusable code is the most convenient tool to speed up the implementation procedure. For the domain of distributed environment, the APIs and idioms play the role of building blocks to deal with most common problems. Those components are neither conceptual models like regular design patterns for developers to refer to nor black boxes like mature frameworks for configuring and customizing. Rather, the solution of GreatFree turns out to be constructible and revisable high quality resources for programming. For that, developers are required to be familiar with the relevant knowledge to come up with a distributed system rather than to follow a tutorial only and forget about the critical technical issues. A high quality, creative and flexible system cannot be implemented unless programmers understand the essences and details. Meanwhile, developers' effort is still lowered with the APIs and idioms. Inspired by developers' knowledge, the components can take the role to program a system conveniently in terms of their high readability, constructability and changeability.

3.1. Specific Code is the Silver Bullet

Specific code is the silver bullet to speed up the development of a self-contained system compared with other resources like design patterns and frameworks. Although they cover common solutions in either the manner of object-oriented programming [17] or even in distributed environments [18], design patterns belong to conceptual models for references rather than the ones that can be embedded into an existing system directly for execution. On the other hand, a mature framework is an established system for customization instead of a self-contained one in which each detail is visible to developers.

Specific code is the reusable program that is proved to resolve one typical issue in a high quality state in a particular computing environment. Once if a bunch of such code is available, it is convenient for developers to either weave them to their own programs following straightforward patterns or reconstruct them to implement other systems further. During the procedure, developers are only required to transform the code to their own particular programs in accordance with the specific code as samples. For the maturity of the specific code, the transforming is similar to a mapping process rather than the conventional one to learn requirements, propose solutions, design and test programs. Developers take the task without considering the details of code themselves in most cases such that the effort is apparently lower than programming from scratch.

3.2. Knowledge of Distributed Systems is Required

GreatFree provides a series of APIs and idioms to assist developers to implement a self-contained distributed system rapidly. Different from the transparency perspective of the traditional software engineering, GreatFree developers are required to learn the specific issues of the system they are implementing. Although it is unnecessary to resolve the problems by programming themselves, they ought to be aware of what components are needed, where to place them in the program and what effects the components take. As a common sense, it is unreasonable that developers know little about the principles of the system they are implementing. As a matter of fact, the programming environment supported by GreatFree is not a black box to hide details from programmers. Instead, it is the composing-enabled open source for them to follow and speed up the distributed system development. Only then, they are able to understand what they are doing such that it is possible for them to raise the quality of their systems and improve GreatFree through revising the open source.

3.3. Changeable APIs and Idioms

GreatFree is an open source development tool that consists of a series of APIs and idioms. Although it is tested, it does not mean that the exact solutions are not replaceable and changeable. With the support of the idioms, the overall structure of the system is stable and developer-friendly. However, it allows programmers to update the internal algorithms, i.e., the APIs, without ruining the code organization upon GreatFree idioms. For example, developers could upgrade the issues like resource management, thread management, multicasting approaches, remote eventing and pulling and so forth. In addition, although the system implemented by GreatFree is scalable, it does not guarantee that it fits in all of the heterogeneous distributed environments. It encourages developers to follow the current version and make changes for their own requirements in either the APIs or the idioms.

4. THE GREAT FREE APIS

GreatFree is comprised of two portions, the APIs and the design patterns as idioms. GreatFree APIs attempt to provide developers with numerous mature encapsulated programs to resolve most distributed problems. The portion of GreatFree APIs covers five areas, including remote interactions, resource reusing, concurrency implementation and control, multicasting and some other utilities. It helps developers construct their systems without taking care about the details of each solution.

4.1. Remote Interactions

The APIs for remote interactions aim to support developers to accomplish four types of remote interactions tasks, including establishing remote connections, sending synchronous/asynchronous notifications, performing remote reading and managing remote IO resources. The APIs for remote interactions consist of 15 classes, as shown in Table 1.

Table 1. Great Free Remote Interactions.

API	Description
AsyncRemoteEventer	This class aims to send notifications to a remote server asynchronously without waiting for responses. The sending methods are nonblocking.
Eventer	This is a thread derived from NotificationObjectQueue. It keeps working until no objects are available in the queue. The thread keeps alive unless it is shutdown by a manager outside.
EventerIdleChecker	The class works with AsyncRemoteEventer to check whether an instance of Eventer is idle long enough so that it should be disposed.
FreeClient	This is a TCP client that encloses some details of TCP APIs such that it is convenient for developers to interact with remote servers. Moreover, the client is upgraded to fit the caching management.
FreeClientCreator	The class contains the method to create an instance of FreeClient by its IP address and port number. It extends the interface of Creatable and it is used as the resource creator in the RetrievablePool.
FreeClientDisposer	The class implements the interface of Disposable and aims to invoke the dispose method of an instance of FreeClient to collect the resource. It is used a resource disposer in RetrievablePool.
FreeClientPool	The pool, RetrievablePool, is mainly used to manage the resource of FreeClient. Some problems exist when instances of FreeClient are exposed outside since they might be disposed inside in the pool. It is a

	better solution to wrap the instances of FreeClient and the management on them. The stuffs should be invisible to outside. For that, a new pool, FreeClientPool, is proposed.
IPNotification	This is an object to contain the instance of IPPort and the message to be sent to it. It is used by the class of Eventer in most time.
IPPort	The class consists of all of the values to create an instance of FreeClient. For example, it is the source that is used to initialize resources in RetrievablePool.
OutMessageStream	The class consists of the output stream that responds a client. The lock is used to keep responding operations atomic. The request is any message that extends ServerMessage.
RemoteReader	The class is responsible for sending a request to the remote end, waiting for the response and returning it to the local end which sends the request.
ServerIO	The class encloses all the IO required details to receive and respond a client's requests.
ServerIORegistry	The class is used to keep all of the ServerIOs, which are assigned to each client for the interactions between the server and the corresponding client. This is a management approach for those instances of ServerIOs.
ServerListener	The class acts as the listener to wait for a client's connection. To be more efficient, it involves a thread pool and the concurrency control mechanism in its internal mechanism.
SyncRemoteEventer	The eventer sends notifications to remote servers in a synchronous manner without waiting for responses. The sending methods are blocking.

4.2. Resource Reusing

Resource reusing is a critical topic since a distributed system consists of some critical resources to be saved. In Java SE, it claims that memory is maintained by the underlying platform. However, other resources have to be taken care by developers themselves. GreatFree includes a bunch of APIs in 16 classes to ease the task, as listed in Table 2. They are roughly divided into three categories, i.e., pooling, caching and relevant interfaces.

Table 2. GreatFree Resource Reusing.

API	**Description**
Creatable	This is an interface to define a resource creator that initiates an instance of the resource in the resource pool, such as RetrievablePool. The resource derives from FreeObject.
Disposable	The interface defines a method for the disposer that collects the resource in the resource pool, such as RetrievablePool. The resource derives from FreeObject.
FreeReaderIdleChecker	The class is used to call back the method of checkIdle of the instance of FreeReaderPool.
FreeReaderPool	This class is similar to RetrievablePool. In fact, it is a specific version of RetrievablePool. First, the resource managed by the pool is FreeClient. Second, it aims to initialize instances of FreeClient for not only output but also for input.
HashCreatable	The interface defines the method to create instances that extend HashFreeObject.
HashDisposable	The interface defines the method to dispose the objects that are derived from HashFreeObject.
IdleChecker	The idle checker works with the ResourcePool to check the idle

	states of resources.
MulticastMessageDispo ser	This is an implementation of the MessageBindable interface. It is usually used by threads that share the same multicast messages.
QueuedIdleChecker	The class aims to check periodically whether a resource is idle for a long enough period. If so, the resource needs to be disposed.
QueuedPool	The pool aims to manage resources that are scheduled by their idle lengths. The one that is idle longer has the higher probability to be reused than the one that is idle for a shorter period.
ResourceCache	This class is a cache to save the resources that are used in a high probability. It is designed since the total amount of data is too large to be loaded into the memory. Therefore, only the ones that are used frequently are loaded into the cache. It is possible that some loaded ones are obsolete. It is necessary to load new ones that are used frequently into the cached and save the ones that are out of date into the database or the file system persistently.
ResourcePool	The pool aims to manage resources that are scheduled by their idle lengths. The one that is idle longer has the higher probability to be reused than the one that is idle for a shorter period. Different from QueuedPool, this pool does not care about the type of resources. It assumes that all of resources in the pool are classified in the same type.
RetrievableIdleChecker	The class runs periodically to check whether a resource being managed in a resource pool, such as RetrievablePool, is idle enough time. If so, it collects the resource.
RetrievablePool	The class is a resource pool that aims to utilize the resources sufficiently with a lower cost. The pool is usually used for the resource of FreeClient. For each remote end, multiple FreeClients are initialized and managed by the pool. When it is necessary to interact with one remote end, it is convenient to obtain a FreeClient by the key or the initial values, i.e., the IP address and the port, which represent the remote end uniquely. That is why the pool is named the RetrievablePool.

4.3. Concurrency Implementation and Control

The issue of concurrency implementation and control is particularly critical to a distributed system since it is required to deal with potential accessing from a huge number of users and other remote clients. GreatFree contains rich solutions to the problem. 35 classes are designed to provide developers with a convenient environment to program a high concurrency and low cost distributed system. Table 3 lists the APIs and simple descriptions of them.

To implement the concurrency, two primary situations need to be dealt with. The first one is to receive concurrent notifications and the second one is to receive concurrent requests and then generate responses concurrently. In details, each of them needs to take into account the issue of multicasting. That is, the notifications and requests/responses are performed within a large-scale environment, which consists of numerous nodes, rather than between two nodes. It proposes a couple of threading queues that encloses incoming messages and dispatchers to manage those threads for specific cases.

For the issue of concurrency control, first of all, it is embedded into the concurrency APIs. Moreover, a new class, Collaborator, which encloses locking and waiting/notifying mechanisms of Java SE, is proposed for developers to use conveniently.

Table 3. GreatFree Concurrency Implementation and Control.

API	**Description**
AnycastRequestDispatcher	This is a class that enqueues requests and creates anycast queue threads to respond concurrently. If the current host does not contain the requested data, it is necessary to forward the request to the host's children.
AnycastRequestQueue	This is a thread that possesses a request queue and other remote communication resources. It is the base one to support implementing anycast requests in a concurrent way.
AnycastRequestThreadCreatable	This is an interface to define the method to create a thread for processing anycast requests concurrently.
BoundBroadcastRequestDispatcher	This is a dispatcher to manage broadcast request threads that need to share requests. The threads must be synchronized by a binder.
BoundBroadcastRequestQueue	When processing broadcast requests, no matter whether the current host contains the matched data, it is required to forward the request to its children. That is the difference between the anycast and the broadcast. However, it is suggested that the retrieval and data forwarding can be done concurrently and they do not affect with one another. The thread is designed for the goal since they are synchronized once after both of them finish their critical tasks. Therefore, they do not affect each other.
BoundBroadcastRequestThreadCreatable	The interface defines the method to create the bound broadcast request thread.
BoundNotificationDispatcher	This is a dispatcher to manage threads that need to share notifications.
BoundNotificationQueue	The thread is different from NotificationQueue in the sense that it deals with the case when a notification is shared by multiple threads rather than just one. Therefore, it is necessary to implement a synchronization mechanism among those threads.
BoundNotificationThreadCreatable	The interface defines the method to create the instance of BoundNotificationQueue. It is managed by BoundNotificationDispatcher.
BroadcastRequestDispatcher	This is a class that enqueues requests and creates broadcast queue threads to respond them concurrently. If the current host does not contain the requested data, it is necessary to forward the request to the host's children.
BroadcastRequestQueue	This is a thread that possesses a request queue and other remote communication resources. It is the base one to support implementing broadcast requests in a concurrent way.
BroadcastRequestThreadCreatable	This is an interface to define the method to create a thread for processing broadcast requests concurrently.
CheckIdleable	This is an interface to define the signatures of two methods for a thread's idle checking.
Collaborator	The class encloses locking and notify/wait APIs to help developers control concurrency.
Consumable	The interface defines the method for a consumer in the producer/consumer pattern.
ConsumerThread	This is an implementation of the pattern of

	producer/consumer.
Dispatchable	It defines some interfaces that are needed in ServerMessageDispatcher.
Interactable	The interface defines a few method signatures for the interaction between a caller and a callee in a concurrent environment. The caller notifies the callee by calling the methods provided by the callee such that the callee can respond to the caller. The caller does that when its running circumstance is changed in a certain situation.
InteractiveDispatcher	This is a task dispatcher that schedules tasks to the special type of threads derived from InteractiveQueue. For the distinct designs, instances of managed threads can interact with the dispatcher for high quality management.
InteractiveQueue	This is the base class that must be derived to implement a thread that holds the methods that can be called to notify the interactive dispatcher.
InteractiveThreadCreatable	In general, a pool needs to have the ability to create instances of managed resources. The Creatable interface is responsible for that. The interface defines the method to create instances of InteractiveThread, which is derived from InteractiveQueue.
MapReduce	This is a high concurrency processing class. Numerous threads that take task queues are the input of the class. It is able to execute those threads concurrently and merge the results from them together.
MapReduceQueue	This is a thread to implement the mechanism of map/reduce.
MessageBindable	Some behaviors, such as disposing, on the messages must be synchronized among threads. If no synchronization, it is possible that a message is disposed while it is consumed in another one. The interface defines the relevant method signatures.
MessageProducer	This is a producer/consumer pattern class to input received messages into a concurrency mechanism, the server dispatcher, smoothly.
NotificationDispatcher	This is a class that enqueues notifications and creates threads to process them concurrently. It works in the way like a dispatcher. That is why it is named.
NotificationObjectQueue	This is a fundamental thread that receives and processes notifications as an object concurrently. Notifications are put into a queue and prepare for further processing. It must be derived by sub classes to process specific notifications.
NotificationQueue	This is a fundamental thread that receives and processes notifications in the form of messages concurrently. Notifications are put into a queue and prepare for further processing. It must be derived by sub classes to process specific notifications.
NotificationThreadCreatable	This is an interface defines the method signature to create the instances of NotificationQueue.
RequestDispatcher	This is a class that enqueues requests and creates threads to respond them concurrently. It works in the way like a dispatcher. That is why it is named.
RequestQueue	This is a thread that receives requests from a client, puts those messages into a queue and prepares for further processing. It

	must be derived by sub classes to provide the real responses for the requests.
RequestThreadCreatable	This is the interface to define a method signature that creates a thread to respond users' requests.
Runner	This is a class to simplify the procedure to invoke a single thread that implements the interface of Runnable of Java SE.
ServerMessageDispatcher	This is the base of a server message dispatcher. All of the messages sent to the server are dispatched by the class concurrently.
Threader	This is a class to simplify the procedure to invoke a single thread that is derived from the class of Thread of Java SE.
ThreadIdleChecker	This is a callback thread that runs periodically to call the idle checking method of the thread being monitored.

4.4. GreatFree Multicasting

To implement a large-scale distributed system, it is indispensible to employ high efficient multicasting mechanisms. GreatFree provides developers with 23 classes to achieve the goals. Table 4 lists all of the APIs. In the current status, GreatFree supports a tree-based multicasting. All of the nodes are organized as a tree to transmit data as messages or objects. The multicasting is divided into the notification one and the request/response one. For the later case, it is further categorized into the one of broadcast request/response and the one of anycast request/response.

Table 4. GreatFree Concurrency Implementation and Control.

API	**Description**
AnycastRequest	The request is a multicast one that is sent to all of the nodes in a cluster. However, once if one node at least responds the request positively, the multicast requesting is terminated. That is the difference from the broadcast requesting.
AnycastResponse	The message is an anycast response to be responded to the initial requester after retrieving the required data.
BroadcastRequest	The message is a broadcast request to be sent through all of the distributed nodes to retrieve required data. For multicasting is required, it extends ServerMulticastMessage.
BroadcastResponse	The message is a broadcast response to be responded to the initial requester after retrieving the required data.
ChildMessageCreatable	The interface defines the method that returns the message creator to generate multicast messages to children nodes in the multicasting topology. It is used to define the instance of children multicastor source.
ChildMulticastMessageCreatable	The interface defines the method to create a multicast message on a child node rather than the root one.
ChildMulticastor	This is the multicasting class to run on a child node in the multicasting topology.
ChildMulticastorSource	The class contains all of the initial values to create an instance of ChildMulticastor. That is why it is named ChildMulticastorSource. It is used by the resource pool to manage resources efficiently.
ObjectMulticastCreatable	The interface defines the methods to create multicast messages to be sent.
RootAnycastReaderSource	This class assists the resource pool to create instances of anycast readers. Thus, it contains all of required arguments to do that.

RootAnycastRequestCreatable	The interface defines the methods to create requests in the anycastor.
RootAnyRequestCreatable	The interface returns an instance of the anycast request creator. It is employed by the instance of RootAnycastReaderSource to provide the method for the resource pool to manage anycastors.
RootBroadcastReaderSource	This class provides the resource pool with initial values to create instances of broadcast readers. That is, it is a class that contains all of required arguments to do that.
RootBroadcastRequestCreatable	The interface defines the methods to create requests in the broadcast requestor.
RootBroadRequestCreatable	The interface returns the broadcast request creator. It is employed by the instance of RootBroadcastReaderSource to provide the method for the resource pool to manage broadcastors.
RootMessageCreatable	The interface defines the method that returns the message creator to generate multicast messages. It is used to define the instance of multicastor source.
RootMulticastorSource	The class contains all of the initial values that are required to create an instance of RootObjectMulticastor. The source is needed in the multicastor pool.
RootObjectMulticastor	The code is a core component to achieve the multicasting among a bunch of nodes. The nodes are usually organized into a particular topology to raise the multicast efficiency. In the case, a tree is constructed for the nodes. For different situations, a more appropriate topology can be selected by programmers.
RootRequestAnycastor	This class is the implementation to send an anycast request to all of the nodes in a particular cluster to retrieve data on each of them. It is also required to collect the results and then form a response to return the root. However, only one response is good enough for anycast.
RootRequestBroadcastor	This class is the implementation to send a broadcast request to all of the nodes in a particular cluster to retrieve data on each of them. It is also required to collect the results and then form a response to return the root.
ServerMessage	The class is the base for all messages transmitted between remote clients/servers.
ServerMulticastMessage	This is the base class to implement the message that can be multicast among a bunch of nodes.
Tree	It aims to construct a tree to raise the quality of multicasting. The tree in the case is simple, such as each node having an equal number of children. It is acceptable when all of the nodes have the similar computing capacity and most of them run within a stable computing environment. For a heterogeneous environment, a more complicated tree or other topologies must be applied.

4.4. GreatFree Utilities

GreatFree provides some utilities to assist the programming for sorting, file input/output, timing, XML and so forth. Although those APIs do not contribute to the distributed programming directly, it is useful in their specific cases, as shown in Table 5.

Table 5. GreatFree Utilities.

API	Description
CollectionSorter	The class aims to sort a collection, a list or a map, in the ascending or descending manner. It is also able to select the maximum or minimum value from the collection.
FileManager	The class provides some fundamental file operations based on the API from File of JDK.
FreeObject	The class is designed in the system to fit object reusing, caching and so on.
HashFreeObject	This is another general object that defines some fundamental information that is required to manage in a pool. Different from the one, FreeObject, this object is managed by the hash key rather than the object key.
NodeID	This singleton is used to save a node's unique ID only.
NullObject	The class represents nothing. It is used when an object needs to fill the placeholder of generics, but it does not matter what should be put there.
Prompts	It contains prompting messages on screen.
Rand	This code is used to generate different types of random number in integer, float and double by enclosing the one, Random, in JDK.
StringObj	This is an object that can be compared by its key in String.
Symbols	The class defines some frequently-used symbols.
TerminateSignal	The class is a flag that represents whether the node process is set to be terminated or not. For some long running threads, they can check the flag to stop their tasks immediately.
Time	The class consists of some common constants and methods to process timing values.
Tools	The class contains some methods that provide other classes with some generic services.
UtilConfig	The class keeps relevant configurations and constants of the solution.

5. THE GREATFREE IDIOMS

Another portion of GreatFree is the design patterns as idioms. The design patterns help developers compose those APIs together in a certain structure to handle specific problems. In essence, they are regarded as the sample code for developers to follow. Because of the maturity of idioms, programming distributed systems become a lower cost task than doing that from scratch.

5.1. Terminator

The idiom of Terminator, which is implemented by a singleton, encloses a flag that represents whether an entire process is terminated or not. It is usually embedded into a long running thread, which detects the flag within a loop or periodically. When the flag is set to the state of being terminated, the thread is ended before the next step of the loop.

At the entry of the entire process, a while loop is designed in the way similar to the above manner. The loop does nothing but sleeps for a certain period in each step. The loop exits after the flag is set to the state of being terminated. The primary API the idiom uses is the TerminateSignal in the GreatFree utility.

5.2. Coordinator

For a large-scale distributed system, a centralized Coordinator is required to manage the entire system for the system initialization, the resource registry, the task and memory distribution, the message bus, the resource disposal and so forth. It is also implemented in a singleton. It contains two primary methods, start() and stop(). Thus, the coordinator is the entry and the exit of a distributed node.

5.3. Server Listener

The idiom of Server Listener is a component that plays the role the remote connection listener as the TCP (Transmission Control Protocol) [1] and accomplishes relevant initialization tasks. For the various goals, it contains the TCP ServerSocket [1], an instance of ThreadPool, ServerIO and its registry, ServerIORegistry. The ServerSocket is used to accept remote TCP connections. When a new connection is constructed and the upper limit is not reached, an instance of ServerIO is created and kept in the registry. Then, the ThreadPool starts a thread to run the newly created ServerIO such that the remote client is served with a concurrency mechanism. In addition, if a peer-to-peer distributed model needs to be implemented, a remote client pool, FreeClientPool, is also taken into account to put in the listener. It needs to initialize an instance of FreeClient for the incoming remote node. For a large-scale distributed system, a peer-to-peer architecture is flexible in terms of implementing highly efficient interactions. Incidentally, each listener maps to one particular port number uniquely. To raise the performance, multiple threads are required to monitor the port. It is initialized and disposed in the Coordinator.

5.4. Remote Eventer

The idiom of Remote Eventer is responsible for sending notifications to a remote node at any moment. As a general eventing mechanism, the notification processes are performed in either a synchronous or an asynchronous manner such that both of the APIs, SyncRemoteEventer and AsyncRemoteEventer, need to be enclosed. In a specific application, all of the eventers are collected in a singleton for convenience. An instance of ThreadPool is also required in the singleton to initialize the asynchronous eventers. It is initialized and disposed in the Coordinator.

5.5. Remote Reader

The idiom of Remote Reader supports another manner to interact with a remote node. It sends a request and waits until a response is received. As a generic module, a singleton API, RemoteReader, is implemented in GreatFree APIs to accomplish the task. For a specific distributed node, it is suggested to enclose all of requests and their calls into a singleton for convenience. It is initialized and disposed in the Coordinator.

5.6. Server IO

Each distributed node needs to derive the API of ServerIO to establish the fundamental input/output streams as a server. A while-loop is needed in the derived idiom and the loop is exited until the remote node closes the connection. In the loop, remote messages are received and then they are assembled with the output stream and the locking for further processing. In the procedure, it employs the API of OutMessageStream.

5.7. Message Producer

The idiom of Message Producer is responsible for delivering messages to the idiom of Server Message Dispatcher for concurrently processing. It is derived from general class, MessageProducer, in GreatFree APIs. Additionally, it needs to be implemented with an instance of ServerMessageDispatcher. To keep concurrent, an instance of Threader is also needed to start the idiom. As required by the Threader, a disposer is defined for the instance of ServerMessageDispatcher. For any distributed applications, only one message producer is needed to dispatch received message for further processing. Thus, a singleton is required to wrap the instance of MessageProducer. When a message is received by the Server IO, it is forwarded to the Message Producer after being assembled with the output stream and the locking using OutMessageStream. The idiom is initialized and disposed in the Coordinator.

5.8. Server Message Dispatcher

Derived from the API, ServerMessageDispatcher, the idiom of Server Message Dispatcher consists of all of the thread management pools to deal with incoming messages concurrently. According to the types of those messages, the thread management pools are divided into different groups for notifications, requests, multicast notifications, anycast requests, broadcast requests and so forth. For each type of messages, it is possible to define some exactly specific sub types of messages, such as the request for sign-in or the notification for online. If the sub types of incoming messages are rich, it is allowed to define multiple Server Message Dispatchers and initialized in the Message Producer. Furthermore, it needs to create a specific thread management idiom for each sub type message. All of them are enclosed, initialized and shutdown in one particular Server Message Dispatcher. The typical structure for the idiom is a switch statement that dispatches incoming messages to each of the thread management idiom according to the message identifications.

5.9. Notification Queue

The idiom of Notification Queue is derived from the API, NotificationQueue, with a specific notification message. In the idiom, one two-level nested while-loop is needed to enclose the operations, which are executed immediately after receiving one notification. The outer while-loop detects whether the thread is shutdown. The inner while-loop detects whether the queue to keep notifications in the idiom is empty or not. If the queue is empty, the thread does not terminate immediately. It needs to be waiting for a period such that it avoids high CPU usages for the long-running loop and it also reduces the cost to create a thread when new notifications are received. Usually, it is a high cost solution to collect the thread immediately when the queue is empty. To save memory, it is also suggested to dispose notifications after it is processed. For that, developers are required to invoke the disposing method.

5.10. Notification Object Queue

The idiom of Notification Object Queue is derived from the API, NotificationObjectQueue. Its structure is identical to that of the Notification Queue. The only difference is that data in the queue is derived from the Java base class, object, whereas the queue in the Notification Queue keeps data derived from ServerMessage of GreatFree. For that, the Notification Object Queue is usually not used as remote concurrent processing unless the object implements the interface of serializable of Java.

5.11. Bound Notification Queue

When a notification needs to be handled by multiple threads concurrently after it is received, it needs to use the idiom of Bound Notification Queue to define them if the notification is probably updated in any one of them. The idiom is derived from the API, BoundNotificationQueue, in which an interface, MessageBindable, should be implemented to enclose the operations to be synchronized.

5.12. Request Queue

As a child of the API, RequestQueue, the idiom of Request Queue has the similar structure to that of Notification Queue, such as the two-level nested while-loop, the message queue, the waiting control to save resources and the message disposing methods. Besides those, a responding method needs to be invoked in the inner loop immediately after the response is created.

5.13. Bound Broadcast Request Queue

The idiom of Bound Broadcast Request Queue is another case to send a request and receive a response from a remote system that is made up with a numerous distributed nodes. It is derived from the API, BoundBroadcastRequestQueue. Different from that of the Request Queue, the request is transmitted to each node in the system rather than to a single one. Additionally, since the request is handled by two concurrent threads, i.e., one for generating the response and another for forwarding it continually, and both of them might change it in some cases if applicable, such as disposing the request, it is necessary to synchronize the relevant operations. That is why the idiom is named Bound. The synchronized operations are enclosed in the unique instance of a class implementing the interface of MessageBindable.

5.14. Anycast Request Queue

Different from that of Bound Broadcast Request Queue, it is not required to get responses from each of them when a request is sent to a large number of potential distributed nodes. If anyone of them responds, the entire procedure is terminated. Derived from the API, AnycastRequestQueue, the idiom of Anycast Request Queue takes the task. It notes that it is impossible that the request is handled by the two operations concurrently, i.e., the one forwarding it and the one responding it. Because the forwarding is probably terminated in anycast, it is not necessary to synchronize the above two threads since the two operations are performed in a sequential order in the idiom. Thus, the idiom is not named as Bound as what it does on that of Bound Broadcast Request Queue.

5.15. Root Multicastor

Being situated at the coordinator node of a distributed system, the Root Multicastor is an idiom for multicasting in the system that consists of numerous distributed nodes. By default, the multicasting is performed within a tree topology and the coordinator plays the role of the root. It is a singleton that includes a bunch of multicastor resource pools, which are implemented by the API of ResourcePool. Each of the pools is able to create instances of specific multicastors derived from RootObjectMulticastor. They send data to all of the nodes in the tree. Each of the methods of the Root Multicastor contains four components, i.e., data to be transmitted, the initialization of a multicastor, the transmission invocation and the collection of the multicastor for reuse. The idiom is responsible for sending notifications only.

5.16. Child Multicastor

The partner of the Root Multicastor is the Child Multicastor, which is located at the children nodes of the multicasting tree. Each of the child multicastors is derived from ChildMulticastor to send data its immediate children to keep the multicasting go ahead. Except that, the structure of the Child Multicastor is identical to the root one. The multicastors are also managed by the instances of ResourcePool.

5.17. Root Anycast Eventer

Sometimes it is necessary to send notifications to some of the nodes in a system instead of all of them. In this case, the Root Anycast Eventer takes the responsibility. It is also situated at the root of the multicasting tree. There are no any differences from the structure of the Root Multicastor. The RootObjectMulticastor is the parent API of each multicastor. It only needs to send notifications to its immediate children.

5.18. Child Anycast Eventer

As the partner of the Root Anycast Eventer, the Child Anycast Eventer keeps sending notifications to its immediate children. But the operation is terminated if the notification is needed by the local node. The judgment is carried out by the idiom of Notification Queue, which receives the notification. The structure of the idiom is identical to that of the Child Multicastor. Each multicastor is also derived from ChildMulticastor and managed by the instances of ResourcePool.

5.19. Root Broadcast Reader

Sometimes, it is required to query all of the distributed nodes in a system. In the case, the idiom of Root Broadcast Reader is competent with the task. Different from those in the Root Multicastor, the multicastors are derived from RootRequestBroadcastor. Besides the multicastor pools implemented by ResourcePool, the idiom needs to implement a mechanism to collect the responses. That is, until the number of responses reaches to that of the total of the distributed nodes, the method of the broadcast requesting is blocked.

5.20. Child Broadcast Reader

As the partner of the Root Broadcast Reader, the Child Broadcast Reader is easier to be implemented since it is unnecessary to take into account any additional issues. Its primary task is the same as the Child Multicastor. The change happens in the relevant threads, i.e., the Bound Notification Queue and the Bound Broadcast Request Queue, to process the broadcast requests. The first queue forwards the requests to its immediate children and the second one creates the relevant response to send it back to the root.

5.21. Root Anycast Reader

Different from the Root Broadcast Reader, another multicast requesting is accomplished by the idiom of Root Anycast Reader if one response at least is received by the root. It is unnecessary to send the request to all of the nodes in the distributed environment if one response at least is received. Similar to the Root Broadcast Reader, the idiom needs to implement a response collecting mechanism besides the multicastor pools. But it only needs to obtain one response and then returns it to the requestor instead of waiting for all of the requests from each node in the system.

5.22. Child Anycast Reader

The partner of the Root Anycast Reader is the Child Anycast Reader. The request from the root needs to be forwarded to the children of the current local node if the local node cannot generate the required response. The implementation of the idiom is identical to the Child Multicastor. The difference happens in the threads that invoke the methods of the idiom. In the case, the corresponding thread idiom is the Anycast Request Queue.

5.23. Interactive Queue

Different from traditional threads, the idiom of Interactive Queue contains a number of callback methods such that the running threads are able to interact with the pool that manages them. The interactions are realized through invoking the callbacks. According to the interactions, it is possible for the pool to notice the status of those threads such that it can make a proper decision to manage those threads effectively and efficiently.

5.24. Map Reducer

The idiom of Map Reducer is an important idiom to implement the high concurrency mechanism. When a large number of parallel tasks are available to be processed and their results need to be merged, it is highly efficient to put them into the Map Reducer if they do not need to synchronize during the procedure to accomplish each of them. The Map Reducer is made up with the pools to manage the resources of MapReduce. Each of its public methods takes numerous tasks as input and the merged results as return values. Inside the methods, it needs to specify the reducer, initialize the algorithm object and invoke the concurrent mapping and reducing.

6. THE EXPERIMENTAL ENVIRONMENT

GreatFree APIs and idioms were proposed during the procedure to implement a new infrastructure of World Wide Web [11][2]. It believes the social capital [36] is the driver to associate the human capital [37] over the Web. For the understanding, it needs to take a heavy load to program from scratch, including the issues of routing, multicasting, persisting, presenting and so forth. It attempts to build a brand new large-scale heterogeneous information system such that users are able to perform various information accessing behaviors, such as publishing, forwarding, commenting, browsing, navigating, searching and following.

The current version is implemented with Java SE 7. It reaches the total lines of code, 373,181, at the server side. In addition, its client side has 11,662 lines of code, which are implemented on Android 5.0. All of APIs and idioms were proposed during the procedure to implement the server side.

The server side is made up with the coordinator, the crawlers, the data rankers, the publisher rankers, the memory nodes and the interactive nodes. In the current version, except that the coordinator is implemented with a single computer, other types nodes are comprised of multiple computers without the upper limit. As the center of the system, the coordinator is responsible for integrating all of the nodes together, such as distributing tasks, disseminating data, forwarding queries and so forth. Each of the crawlers receives tasks from the coordinator, collects Web pages from the Web and then injects them into the system for sharing in a concurrent manner. The data rankers evaluate and sort data in a certain order such that it can be presented to users in a high quality form. The node rankers are responsible for sorting publishers of data and the consequences are useful to construct the graphs between publishing organizations and individuals. The memory nodes achieve the goal to keep important data in memory as cache and

the rest data on disk for persistence with a distributed cluster. The interactive nodes connect with users' devices directly and respond to users' requests. Each of the above nodes is implemented from scratch with Java SE initially for the proposals of new distributed models and data transmission protocols. For that, the relevant APIs and idioms are proposed after a long-term accumulation. Nowadays the backend of the system is accomplished, but its clients are still under construction. It will be launched as a commercial system in one or two years.

7. FUTURE WORK

As a comprehensive solution to the large-scale distributed system, it can hardly claim the APIs and idioms discussed in the paper cover all of the issues. With the progress of the development of the system, more APIs and idioms must be put forward. In accordance with the domain of the social computing, it is expected to invent some patterns that fit in the unstable environment. In addition, it needs to improve the thread management approaches in the current version since it notes that the number of raised threads is high. One reason for that is due to that each dispatcher has a parent thread that is always alive. For such dispatchers are numerously utilized, it leads to the growth of the number of threads. It can be solved through the lazy initialization and the parent thread is killed after a certain period. Finally, all of the algorithms in each APIs and idioms need to be improved. For the limited time, the algorithms are still rough. For example, the caching and pooling algorithms are to be optimized. Potential developers are able to strengthen those algorithms with the open source.

REFERENCES

[1] Elliotte Rusty Harold. 2014. Java Network Programming. O'Reilly Media, ISBN: 978-1-449-35767-2.

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

[3] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes and Doug Lea. 2006. Java Concurrency in Practice. Addison-Wesley, ISBN: 978-0-321-34960-6.

[4] Joshua Bloch. 2008. Creating and Destroying Objects, Chapter 2, Effective Java. Addison-Wesley, ISBN: 978-0-321-35668-0.

[5] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Architectures, Chapter 2, Distributed Systems: Concepts and Design, the Fifth Edition. Addison-Wesley, ISBN: 0-13-239227-5.

[6] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Consistency and Replication, Chapter 7, Distributed Systems: Concepts and Design, the Fifth Edition. Addison-Wesley, ISBN: 0-13-239227-5.

[7] Jim Farley. 2001. Message-Passing Systems, Chapter 6, Java Distributed Computing. O'Reilly Media, ISBN: 1-56592-206-9E.

[8] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Communication, Chapter 4, Distributed Systems: Concepts and Design, the Fifth Edition. Addison-Wesley, ISBN: 0-13-239227-5.

[9] Ken Arnold, James Gosling and David Holmes. 2005. The Java Programming Language. Addison-Wesley, ISBN: 0-321-34980-6.

[10] George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair. 2011. Processes, Chapter 3, Distributed Systems: Concepts and Design, the Fifth Edition. Addison-Wesley, ISBN: 0-13-239227-5.

[11] Bing Li. 2015. DOI: https://github.com/greatfree/Programming-Clouds.

[12] Tomcat. 2015. Apache Tomcat. DOI: http://tomcat.apache.org/.

[13] Danny Coward. 2015. Java EE 7: The Big Picture. McGraw-Hill Education, ISBN: 978-0-07-183734-7.

[14] Chuck Lam. 2011. Hadoop In Action. Manning Publications, ISBN: 978-1-93518-219-1.

[15] Dan Radez. 2015. OpenStack Essentials. Packt Publishing, ISBN: 978-1-78398-708-5.

[16] Juval Lowy and Michael Montgomery. 2015. Programming WCF Services: Design and Build Maintainable Service-Oriented Systems. O'Reilly Media, ISBN: 978-1-491-94483-7.

[17] Steven John Metsker. 2002. Design Patterns Java Workbook. Addison Wesley, ISBN: 0-201-74397-3.

[18] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. 1996. Pattern-Oriented Software Architecture, A System of Patterns, Volume 1. John Wiley & Sons Ltd., ISBN: 0-471-95869-7.
[19] Douglas Schmidt, Michael Stal, Hand Rohnert and Frank Buschmann. 2000. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2. John Wiley & Sons Ltd., ISBN: 0-471-60695-2.
[20] Michael Kircher and Prashant Jain. 2004. Pattern-Oriented Software Architecture, Patterns for Resource Management, Volume 3. John Wiley & Sons Ltd., ISBN: 0-470-84525-2.
[21] Frank Buschmann, Kevlin Henney and Douglas Schmidt. 2007. Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing, Volume 4. John Wiley & Sons Ltd., ISBN: 978-0-470-05902-9.
[22] Frank Buschmann, Kevlin Henney and Douglas Schmidt. 2007. Pattern-Oriented Software Architecture, On Patterns and Pattern Languages, Volume 5. John Wiley & Sons Ltd., ISBN: 978-0-471-05902-9.
[23] Paul Whitehead, Ernest Friedman-Hill and Emily Vander Veer. 2002. Java and XML. Wiley Publishing, ISBN: 0-7645-3683-4.
[24] Solr. 2015. DOI: http://lucene.apache.org/solr/.
[25] JBoss. 2015. DOI: http://www.jboss.org/.
[26] WebSphere. 2015. DOI: http://www.ibm.com/software/websphere.
[27] FlashGet. 2015. DOI: http://www.flashget.com/index_en.html.
[28] Skype. 2015. DOI: http://www.skype.com.
[29] Twitter. 2015. DOI: http://www.twitter.com.
[30] HTTP. 1996. HTTP – Hypertext Transfer Protocol Overview. DOI: .http://www.w3.org/Protocols/.
[31] JavaServer Pages. 2015. DOI: http://www.oracle.com/technetwork/java/javaee/jsp/index.html.
[32] Ian Clarke, Oskar Sandberg, Brandon Wiley and Theodore W. Hong. 2001. Freenet: A Distributed Anonymous Information Storage and Retrieval System. International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability, 2001.
[33] Matei Ripeanu. 2001. Peer-to-Peer Architecture Case Study: Gnutella Network. Proceedings of 1st International Conference on Peer-to-Peer Computing, 2001, pp. 99-100.
[34] Time-Berners Lee. 1989. The Original Proposal of WWW and HTML. DOI: http://www.w3.org/History/1989/proposal.html.
[35] Bing Li. 2015. The New Infrastructure of WWW. DOI: http://greatfree.blog.163.com/.
[36] Nan Lin. 2001. Social Capital: Capital Captured Through Social Relations, Chapter 2. Social Capital – A Theory of Social Structure and Action, Cambridge University Press, ISBN: 0-521-47431-0.
[37] Nan Lin. 2001. Theories of Capital: The Historical Foundation, Chapter 1. Social Capital – A Theory of Social Structure and Action, Cambridge University Press 2001, ISBN: 0-521-47431-0.