**Assignment Two for CS5223**

**Technique Paper Literature Survey**

|  |  |  |
| --- | --- | --- |
| **Topic of the survey:** | **Server Architecture** | |
| **Length of the survey in number of pages (excluding appendix)** | **Survey report: 15 pages**  **Appendix: 3** | |
|  | **Member #1** | **Member #2** |
| **Team Member’s matric number** | **A0159354X** | **A0091827X** |
| **Team Member’s name** | **Liu YuanCheng** | **Lu Shumin** |

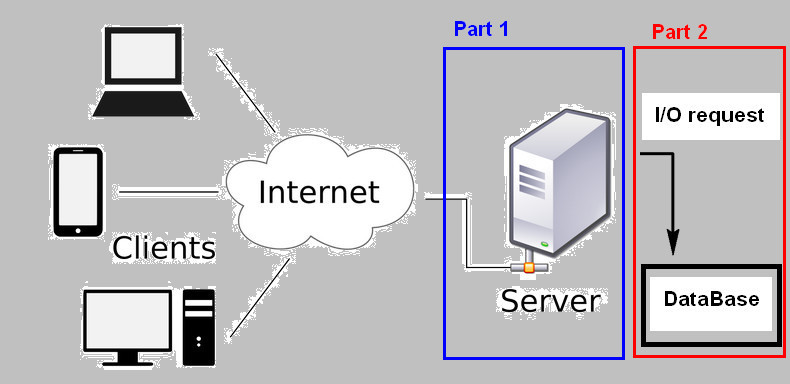
**Topic Introduction**

Introduction of server and client–server model architecture

The modern internet technology connected people together with computers, smart phones and many other devices. For the organization / people / companies who want to provide service through the network, the servers are needed. Generally, a server is a set of programs which are running on a computer/ a distributed computers system to provide functionality to other programs or devices. It is used as the central repository of data, resource and various programs that can share to the users in a network. In the current word, most of the server system are a cluster of computers with some distributed manage program. There are two aspects need to be considered when doing the server technique development: hardware and software. In our research we will focus on the software aspect of the server architecture design.

The most common server model is “client–server model”. In this model, the basic requirement of the client–server model architecture is a single response socket overall computation which is distributed across multiple processes or devices. A server can service significant number of clients with in an acceptable response time, and one single client program can connect multiple servers simultaneously. So the concurrency problem handling, algorithm optimization, performance and fault tolerance improvement are the critical problem need to be considered during the server architecture design.

The diagram of the client–server model is shown below and our research will focus two parts on the right side: one part is the computer cluster in front to handle the request from different clients (provide function and do computing work) and the other part is resource access (data transition, exchange and storage) The architecture looks simple, the server just needs to handle the request from several different devices and process the data I/O request in a user acceptable response time. But after considered the problem introduced in the previous paragraph, to figure out a suitable architecture solution is not an easy work. Based on the algorithm of “Part1”, there are three kinds of server architecture: Multi-threaded architecture, Event-driven architecture and Hybrid architecture.

**-----------------------------------------**

The server architecture in the real word is much complex than the diagram on the left. The file-wall unit, PSTN gateway, switches, backup-recover unit and different kinds of sub-server system are working together as one distributed system.

**-----------------------------------------**

Introduction of server category and architecture design requirement

The server system can be classified to several different kinds of sub-system based on its function. Each sub-server provides its special function to handle one or some service scenarios, so different sub-servers have different architecture. We are focus on five kinds of sub-server system architecture design during doing the paper leaning and research:

1. Application Server: This kind of server is used to host Web-Apps (the program which run in the web browser to the client. The user can run and use this Apps’ function without pre-install a program in their computer) [1] It should provide the advantage of event-driven programming to handle the user continuously requirement during the usage time with an acceptable response time.

2. Catalog Server: It provide a contents index list of information across a large distributed network. This kind of server is used for searching something in the website or on the network. The main requirement of these kinds of servers is high performance information process algorithm and fast I/O data scanning process.

3. Communication Server: Nearly all non-single computer server system has this sub-system. It is used not only to handle the large amount of the user’s service request, but also deal with the data flow/function-call inside the distributed server system. It works as the role of traffic police in the server architecture, such that the multi-threading process and the ability to handle heavy work intensity are required for this kind of server.

4. Computing Server: The main program is running on this server to handle user request, control and manage other sub-servers to finish the tasks. Fast computing speed and program execution efficient is the base demands. For this kind of server, a flexible design, re-useable and extendable functionality are required by the server developer for adding new function, debugging and doing maintenance.

5. Database/File server: These kinds of server are used to handle the I/O, message filter, information save/load problems to help the computing server to finish the jobs. Highly efficiency data exchange and data synchronization between different computers are urgently required.

There are several other “edge” server such as mail server, media server, sound server, proxy server and Web-page server which share one or some aspects of design requirement with the above servers. We will not do discussion on them but in this paper we will pay some attention on the “network game server” architecture and share some experience about how to use the method introduced in the paper we read to improve the Maze-game server architecture which we designed during the Assignment-1. The network game server, especially the multi-player game server is nearly a combination of all the five kinds of server shown above to handle massive concurrency client requirement and keep game data synchronization for all the players.

Introduction of the technique paper view

We reviewed five server architecture papers. They detailed introduced four kinds of server architecture and make a comparison discussion about how to improve the server performance. We briefly do a short introduction about what the five papers focus on:

“SEDA: An Architecture for Well-Conditioned, Scalable Internet Services” [2]

* This paper provides improvement architecture of the event-driven server. It uses explicit queue method and dynamic resource controllers to split the jobs to different stage to balance the whole workload, so the system can perform higher when support massive concurrency demands compared with normal event-driven server under the same hardware condition.

“Capriccio: Scalable Threads for Internet Services” [3]

* Capriccio is a scalable use-level thread package for improving the thread-based server. It is aim to overcome the drawback of thread based server system architecture to provide similar or even higher performance as the event-based server. For the server-thread data sharing and exchange, it introduces the new method “linked stack management” and “resource-aware scheduling” to minimizes the amount of wasted stack space. These method makes data stack grow or shrink at runtime to handle the data assessment and provides a better scheduling and admission control to adapt to the system’s concurrent resource usage.

“Web Prophet: Automating Performance Prediction for Web Services” [4]

* Web-Prophet is focus on the Web-page server design, it employs the “novel” technique to analyses the dependency of the object in the webpages. Then it automates performance prediction for web services to optimize the loading sequence of web pages to the client to highly decrease the page loading time and page error rate.

“Events Can Make Sense” [5]

* This paper introduces another improved event-based server design for managing the concurrency client request for the application server. The program developed by “Tame” can avoid some “stack-ripping” problem for normal event-driven server, it also reduces the runtime overhead and improves the server-program’s execution compatibility on different server platform.(highly decreased the server deployment and update time)

“Why Events Are A Bad Idea (for high-concurrency servers)” [6]

* This paper provides polarity idea about the event-base server with the discussion and experiment. Based on its comparison experiment (It creates its own highly concurrent application) between event-base server and thread-base server architecture, the paper wants to show that a good implement threads-base server can achieve all the strengths of events-server, including support for high concurrency, low overhead, and a simple concurrency model.

After paper reading and research, we found the main technique discussion and argument are between base-driven and the threads-base server. At the pure programming, mathematical and methodology level, a lot of development is done for making one kind of architecture to take the advantage of the other and avoiding the architecture’s own drawback. At the industry and practical usage level people prefer to combine them together to do the hybrid server design as each of the two-kind architecture cannot fully replace the other. (The main idea is similar as the logic in the “SEDA”, designer spit the whole server process and select the architecture based on the stage scenarios) We are also agree that there is not a kind of “Best” architecture which can be applied for all the server design and even the method developed based on convictive experiment data may perform very different in real world. For different kind of server as shown in the previous section, the methodology about why and how to make a good “selection” for different server architecture is also a very valuable aspect of research.

**Problem Specification of Server Architecture**

As introduced in the previous section, clients’ requests are very different amount various servers. Different requests caused different problem in the server architecture design, but normally there are several common problems for most server designer to consider. The five papers discussed some common, but detail problem which designer need to face:

1. Concurrency service requests handling problem.

The frontier problem when servers are joining in the internet is about how to process the massive concurrency demands/request. Some math maturity and algorithms are urgent need for the request service queue to avoid the request jamming or request queue’s buffer overflow.

1. Sub-System coordination and control problem.

The distributed server system looks as a single computer to the outside. But inside the “black-box”, how to arrange the working of different subsystem and control the data flow is a very complex problem for the designer to consider. (Efficiency of communication server)

1. Server data management and exchange problem.

Even a very good arrangement/sub-system management system is designed, without the support of good data exchange and resource usage architecture the system still cannot match the basic server design goal. The problem of how make single computer file save/load, memory access efficiency and how to make multiple computers’ data synchronize is as important as the server control program.

1. Server resource utilization problem.

The computers used to make up the distributed server system have their own hardware limitation. For the server designers, they cannot avoid considering the problem of making the server program running efficiently, increasing the resource utilization and decrease the server overhead. At the same time, computers may crash and the system may get fault during the running time. The server backup, system recovery, service fault tolerance and the aspect of server maintenance are all very important problem to make the server works normally. For “further” utilization, whether the designed architecture is easy for function extension and debugging are all very import.

1. Server response time and delay problem.

The client’s request goes through the server and the response is sent back. The server response time is also a very critical problem. The service need to control their response time under an acceptable time interval and make the service response delay as short as possible. This problem directly affects the user’s feeling of the Web-page server’s or the game server’s service quality.

6. Stack ripping problem.

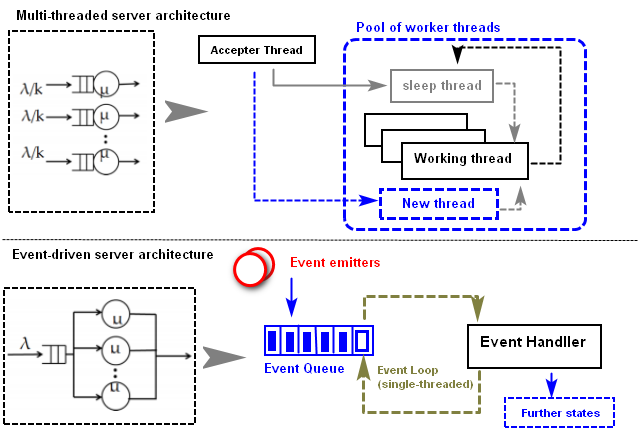
This is a problem programmer may suffer during the design an event-based server. Because multiple tasks share one stack, each function that might block has to been ripped, which will influence this function’s control flow and make memory management complicated.

For the methodology and technology introduced in the 5 papers, the SEDA and Capriccio focus on the problem of concurrency request handling. Capriccio provides some great solution to increase the unification of the system resource and increase the data exchange speed. Tame mainly solve the stack ripping problem, also provides the way to incorporate threads and events together in the same program and automate memory management. “Why Events Are a Bad Idea” give a valuable comparison about the performance of event-driven base server and thread based server when facing the concurrency request handling problem. The Web Prophet provides a flexible solution for decreasing the page load time’s problem, it also highly increase the response time by send the web-page component part by part. (The total response time interval may be only improved a little by the Web-Prophet, but when the user sees the page is load part by part with short period their feeling about the response quality will increase a lot)

**Architecture Technology Summery and Algorithm Analyze**

The five papers introduce several programming technology and mathematic algorithm to solve/reduce the positive influence of the problem in the previous section. We will summarize the working-flow sequence of their technology and do some analyze in this section.

The special function/architecture of a server is demanded by what task it will handle. The 5 papers mainly introduce 4 kinds of server architecture, but the main working-flow sequence overview among them are nearly same. For all kinds of the server, the first step of the server-client request handling process is receiving the user requests which are massive concurrency demands. When the server system is handling the huge number of the request, there are 2 main Service-level agreements: The fist one is the server needs to provide robust performance with a wide range of services; the second is the server need to provide acceptable response time of the service event under huge variations in load. Two main basic technical Architecture is shown below:



Multi-threading Server architecture.

Cons: As shown in the above working-flow diagram, the multi-thread server architecture server’s logic is based on the k M/M/1 queue system. The thread-based approach basically associates each incoming connection with a separate thread/process, each new connection is handled by a dedicated activity. It uses synchronous blocking I/O and handle all task sequentially. It uses the multiple threads and process to achieve the real concurrency. The software structure is simple.

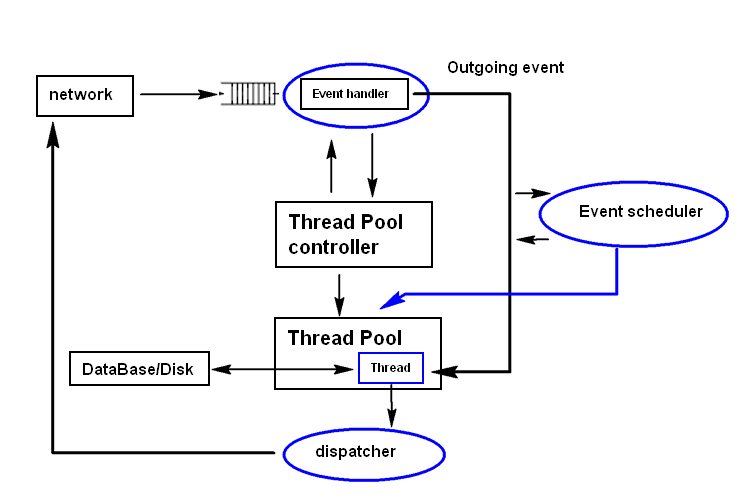
Pros: Under heavy request load, a multi-threaded web server consumes large amounts of memory (due to a single thread stack for each connection), and constant context switching causes considerable losses of CPU time.

Event-based server architecture:

Cons: The thread based server’s logic is based on the M/M/k queue system. Normally the server contents an event Queue to buffer the new incoming event, there is a single threaded event loop do dequeue the event and send the event to event handler. Then the handler will use cascade of asynchronous calls and callbacks that get executed on events. The control flow of an application following the event-driven style is somehow inverted. Normally the utilization performance of the Event based server is better.

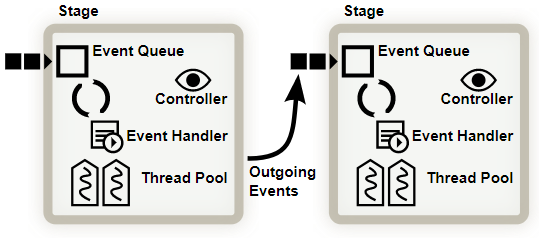
Pros: The notion often makes the flow of control less obvious and complicates debugging.

The staged event-driven architecture (SEDA) is a hybrid approach (combine the two basic architecture) Its can be described as a “stage based” server to manage the concurrency request, I/O scheduling, and resource. It follows the main design of the event-driven architecture (Event driven requirement handling) and use the thread based methodology as the basic programming for the detail computing process. The response time for the request is fast and with multi-threading it can process large amount of concurrency demands at the same time. Its thread pool architecture makes the server hardware utilization efficient and scalable for the huge variations in load which may appear. The main working flow of each state of SEDA is shown below:



Comments: The whole process of client request process is split to several stages. For each stage, when the client request/the previous stages request join in the stages service queue, the event handler will check with the stage’s thread pool control whether the stage system has “free” thread to handle the event, if the thread pool utilization is full the thread pool controller will create new thread for the event and response the event handler. When the event goes out from the event handler, it has already known which thread it will go. The event scheduler will scan the event’s work intensity at the same time, if the workload of the event is heavy, the event scheduler will ask the thread pool to create special I/O thread to help the main event thread to handle the data flow problem to improve the process time during the event handling stage.

An entire work-process for a job with several stages structure is show below. When a stage finished part of the job, the response dispatcher will dispatch the result to the user if it is independent. (Such as the user’s request has 3 separate unrelated question, when the thread finished process the first question, the dispatcher will provide the answer of the first question to the user fist then wait the thread to finish the others, so the response time of the server will be decreased)



The SEDA use two queue (a I/O request Queue and event queue to complete the work ) to implement Resource transparency and Asynchronous access. It use the technology “asyncSocket layer” to share resource across all sockets and do the data process under three stages:

* ListenStage: Operation on [asyncConnection] socket will be converted into a request and placed on request Queue.
* ReadStage: Every other stage can access the resource by put “readlines” operation in the event queue no matter the data socket has available data.
* writeStage: Pick the data write operation from event queue if the data is un-locked.

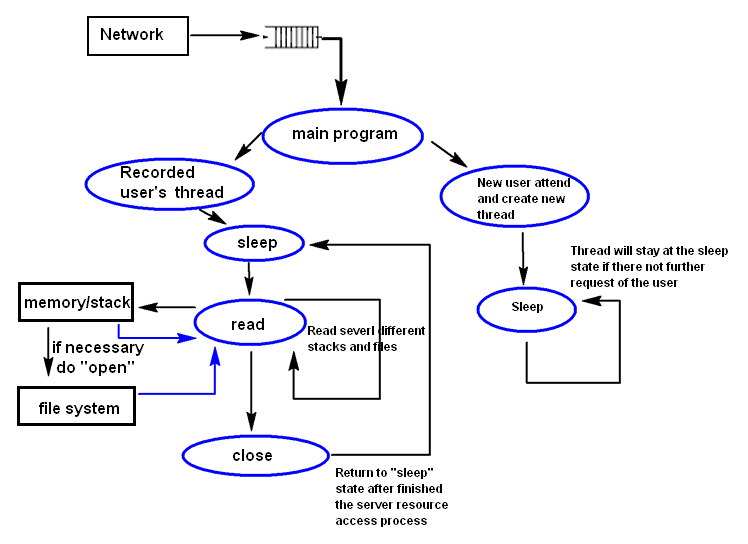
The SEDA is good but not all the client requests are complex and need so much I/O process of the access to the resource. Assume we have thousands of similar simplified clients request and each request only use “less than one” stage to finish: if we still use the SEDA under this scenario, the stage creating time and the computing time of the dispatcher/scheduler is even longer than the time of the computing time. So other pure threading server architecture is introduced in the paper “Capriccio: scalable threads of the internet service”. This kind of server architecture is used to handle a large amount client requests which have slightly light files I/O access or assess the same resource in the server computer system.

Put the shared resource/data with heavy access intensity in memory will highly improve the performance: In Capriccio solution, these shared files will be pre-loaded in the computer’s memory (stack) to increase the resource time. The asynchronous memory[stack] I/O operation needs the OS serial kernel level programming mechanisms.

For the request handling in the user-level programming, Capriccio use loop mechanisms. To overcome the disadvantage of the loop design, Capriccio also provides “Thread Scalability” architecture to monitor the incoming workload in the service queue. According to the paper’s introduction, it can “Improve the number of the thread according to the current require work load. Producers put empty messages into a shared buffer to pre-create the space for the workload increment.” [4] From the experiment data provide from the paper, it shows the user-level thread can greatly reduce the overhead of thread synchronization.

Capriccio handles the data access by two ways. For the data access, dynamic allocation and de-allocation of stack chunk to improving the ability to tune individual application memory usage. For the saved file access, the I/O technology Capriccio used is “linux[‘epoll’]” instead of the using the system call select() and poll() as other system program: it keeps monitors multiple file descriptors to see whether there is I/O operation on them and the epoll consists of a set of user space functions, each taking a file descriptor argument denoting the configurable kernel object.

The work-flow sequence of the Capriccio is shown below, on the memory stack part the memory of guard pages is disable for avoiding unnecessary kernel crossings and virtual memory waste.



Cons: Capriccio extends the advantage of multi-thread server and improves the utilization of the server machine with the kernel level. After read the paper we think its architecture is good to act as a network multi-player game server. Assume the situation: a lot of players are attacking a enemy /monster in the game like WOW, each player sends the “attack” request to the server and the data of the monster in the game(such as the hp, mp and de-buff) should synchronous to every players. The players’ attack request contents very little information (a very short string with few bytes) but the number of the data is huge during every second. We can use Capriccio to implement all the the information of the monster in the memory stack and create a thread for every player who attend in the attacking area. (And for the player who temporary leave the area and will join back, their service thread will go “sleep” and when the player back their service thread will wake up) If the user does some action (attack/add de-buff/ recovery other players) the server will access the data of the bass from the memory directly. If suddenly many users have attend in the area or the user create a lot of request, the controller will increase the thread number to deal with the heavy workload for the next server loops. As stack space is allocated to threads on demand in relatively small increments and is deallocated when the thread requires less stack space, the server machine’s size of virtual memory dedicated to stacks will be significantly decrease.

Pros: When the client concurrency is very low, the time for it to issue system thread call “epoll\_wait()” will make this kind of architecture slower than the event-based single request queue server.

Architecture of web server

After view the normal application server and gamer server technology, we will pay our attention on a kind of server which is closest related to people’s daily life – the Webpage server. The design requirement of the Web-page server is simple: provide the required Web-pages to the user with fast and efficient way. Web-Prophet introduce a new meth methodology “page object dependency detection” and “Prediction of the work load and Automating process adjustment” to make the web server’s performance faster and more intelligent.

For a HTML web pages loaded from the server to the client’s browser(when the client type in the URL or click a link), there are 4 main parts:

* Wireframe structure (The main HTML text document, it determine the whole page structure)
* Function (Such as the javascript, flash plug-in and java-applet handle the user’ request)
* Self-explanatory content (Some document, image, sound file or media content)
* Aesthetics (mostly refer to the CSS of the webpage)[6]

Most of the normal user will not distinguish the above 4 parts clearly; their first consideration and the most important factor they evaluate the service quality is about a Web is its page load times and the page load fault tolerance. Poor page load times (PLT) result in low service usage, which in turn may undermine service income and there are 4 main time element which will make obvious influence of the server’s average page loading time:

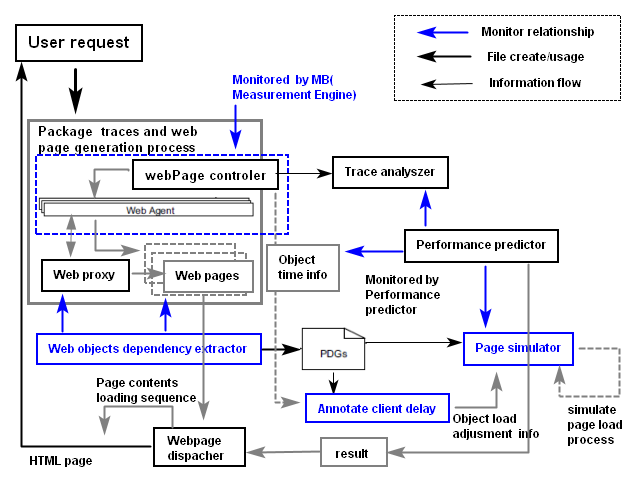
DNS lookup time: This is time to match the DNS in the users typed in URL to the IP address of the web server (As motioned in the lecture, the browser retrieves a resource will reduce this a lot )

Network round trip time (RTT) : Round-trip delay time (RTD) or round-trip time (RTT) is the length of time it takes for a signal to be sent plus the length of time it takes for an acknowledgment of that signal to be received.[7](from wiki)

Server response time: there are several factors which can affect the server response time such as Databases, Traffic and Hosting, Resource Usage and Server Software. In the paper” Automating Performance Prediction for Web Services”, it provide a kind of server architecture to handle these factor and do the improvement.

Client execution time: The Web applications rely on JavaScript or some other client-side scripting will take some time for function handling.

The key idea of Web-Prophet is monitor the dependency of the objected in pages and load them in sequence to the user. Then it simulates the logic to all the similar pages to create an optimized solution which can be used for all the similar web-pages in the future. The work-flow process diagram is shown below(black part is the main work process and the blue part is what Web-Prophet is monitoring and analyzing for the solution generation)



Comments:

There are five main steps in the Web Prophet’s working sequence:

1. Create the sample HTML pages base on the user’s URL request.(No page load template)
2. During the first page creation, use the “Measurement engine” to web network traffic, file I/O, distributed server computer can provide the information to the analyzer.
3. The Dependency extractor uses novel algorithm to infer dependencies between web objects. (The most dominate dependency factor is the download times of individual objects)
4. The performance predictor Implements a simple accurate method to simulate the page load process of a web browser based on the given dependency graph of a webpage.
5. The webpage dispatcher will come a load sequence of the page and the similar pages’ contents to the user base on the dependency graph.

According to the experiment result of the paper, the page load latency is decreased a lot when using the architecture design of Automating Performance Prediction.

The last server system we discuss is Tame, which is an event-based programming library to help solve “stack ripping” problem. In event-based server, since multiple tasks share one stack, programmer has to rip each function that may block, which influence this function’s control flow and complicate memory management. In case this, Tame provides high-level, type-safe APIs for event-based programming that not only solve stack-ripping problem but compatible with legacy event code. Tame also can incorporate threads and events together in the same program and help to manage memory of events.

In the figure below, we compare the function expressing in event with stack ripping problem and the function in events with Tame library. As we see, code B “Simple Tame” saves a lot of codes and improve the readability, compared with code A “Stack-ripped code”. In Tame semantics primitives, there are four related abstractions to hand concurrency: event, wait points, rendezvous, and safe local variables. Event object is allocated by function mkevent with type T. To block functions, wait points can be achieved by “twait {statements; }” or rendezvous objects. Local variables can be sate by enclosing them in “tvars{}” block. By implementing Tame syntactic sugar, code C can be simplified to code D, which is usually same easy as function expressing in thread.

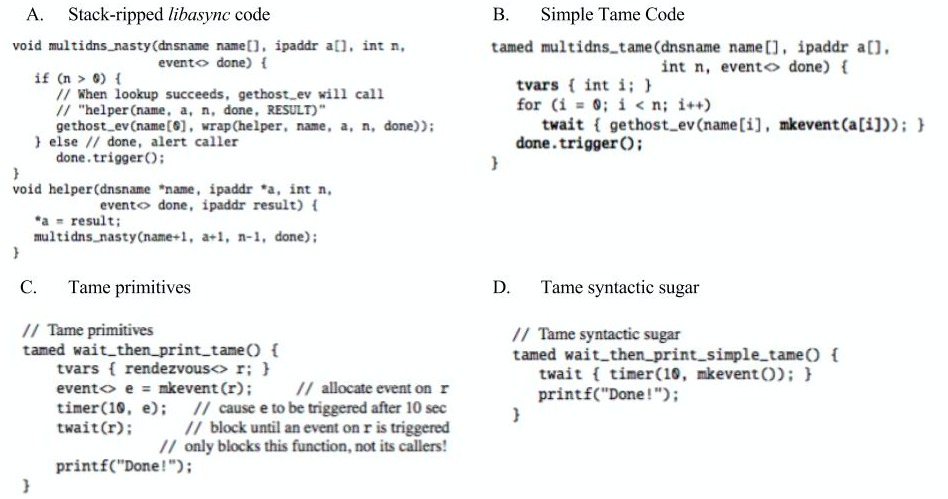


Figure X: Tame sample code

Except event-based programming, Tame also can interoperate with threads when thread is supported. Main difference between event and thread in Tame is the return type of function. Event-based function should use tamed as return type, while thread-based function does not return tamed type. When starting a thread, we need to use tfork() function by passing thread function. When blocking a thread, we use the same wait point syntax twait() as event function. In this case, we can transit between event programming and thread programming easily using Tame library.

Tame provides memory management so that programmer will not suffer from catching memory leaks and wild writes. When using twait{} wait point, Tame can guarantee that memory management is leak-less if syntax is correct. When using rendezvous wait point, the following key invariants must be enforces: after function is finished for the last time, this function can be closed; after all events created in this function are triggered, this function can be closed; before rendezvous r is de-allocated, all associated events must be trigger once. In Tame, these invariants are enforced by two type of reference counting: strong reference and week references. Strong reference is traditional reference counts, while weak reference means one object is accessible only if it has not been de-allocated. This memory management scheme is handled by Tame automatically, which free programmers from garbage collection.

We introduce at the beginning of this section, the thread-based system is base one the k M/M/1 queue theory and the event-driven based system is based on M/M/k queue theory. Normally from the mathematical calculation, will a limited hardware overhead the M/M/k queue will have a better performance. But the fact in real world is different, the paper “Why Events Are A Bad Idea for high-concurrency servers” give a compellentable argument with doing the comparison in three aspects:

In the “Control Flow” part**:** They show that the thread system can express control flow and encapsulate state in a more natural manner when the event systems is suffering stack ripping problem.

In the “Exception Handling and State Lifetime” part: The experiment data proves that the heap allocating and freeing state for the in event system can be difficult. The garbage collection process of event-driven architecture is not as efficiency as the multi-thread architecture for high- performance systems.

In the “System design” part: The thread-based system design is much simpler than the event-driven one.

With the support of their experiment data(they claim the data is accuracy but haven’t reach 100% because of the experiment condition), when under proper implementations the thread based server architecture design can achieve same or very better performance with the event system with a much simpler programming/code-test-debug work intercity.

After read other papers about 4 kind of server architecture, we are convinced double confirm their conclusion. The server architecture design/selection must relate to the category of request: For thread-base, event-driven or hybrid one, there is not a model definitely better than the other, the fact is there is one/some model is more suitable to process some kinds of client request.

**Pros and Cons (Comments)**

After more or less understand the methodology introduced in the paper, we will answer some question and do some discussion related the Maze game server we designed in the assignment1.

Q1. What are the nice properties of these techniques?

SEDA: The stage design which combines the event-driven and multi-thread.

Capriccio: Using memory stack control to improve the efficiency of the shared data utilization and make the thread pool scalable based on the work intensity

Web Prophet: Identify the dependency of the objects in a web-page.

Tame:< >

Currently the Maze gamer server is only 15x15 size with 4 player inside play, if we want to increase the Maze game with a large size such as 500x500 and about server thousand players want to join/leave the game during the day.(At people “free” time there may be several thousand players in the gamer and during the night there are only several hundred players) We will use the property of the Capriccio to create one thread for one user as the user request is very short and all the player as accessing the same shared data.

Q2. What are potentially problems?

We did some introduce at the problem specification part and there small potentially problem for each of each of the techniques:

SEDA: In the SEDA architecture design, for one work, its different work stage may not fixed under one machine (The function/method caller and the callee sometimes are in different program / lib / computer) So during the maintenance and debugging work, the whole-workflow sequence of a job is not viewable for the user/developer, that make them a little difficult to understanding the whole logic, fix some bugs and do the recovery work if the serve crashed.

Capriccio: When the request concurrency is low the system performance is not good. The most important potentially problem is the method need to turn off memory of guard pages, if the system running on the server machine doesn’t allow this, some function of Capriccio may not work properly. What’s more, when the memory page guard is disabled, if the program is not designed properly the memory over flow problem will appear and the hacker will use that as a break point to attack the server. (Especially some program on the server is written by C/C++)

WebProphet: The paper introduced a potentially problem, as part of the design is based on the behavior of browser (IE, Firfox and Chrome) under HTTP/1.1 with HTTP pipelining disabled, if the user use some unconsidered browser (not support dynamic contents) with HTTP pipelining enabled, one slow response in the pipeline will block other subsequence response. Then the whole performance of the server will drop.

Tame:

<>

Q3. How does the technique compare with each other?

The comparison amount event-driven, multi-thread and Hybrid server, SEDA vs Capriccio are all shown in the technology summary and algorithm analyze part.

Q4. Is the technique revolutionarily novel or is it just some minor tweaks on existing techniques?

Base on the reading, SEDA and WebProphet are technique revolutionarily novel. The Capriccio is minor tweaks on existing techniques (Linux system call and stack management technology)

Q5. Are the techniques practically feasible?

Yes, we think they are practically feasible. For example, when we want to extend our assignment Maze game from assignment1 to a huge big 100000 x 100000 one maze. (like some sandbox network multiplayer game) We can separate the whole maze to server small area(1000x1000) and each area is implemented by a server which uses SEDA architecture. And for each server at the user login and playing stage the insider request handling part we can use Capriccio architecture so event the user density is very high on some area the server still can work properly with the acceptable performance. Can use <TAMe> some where?

Q5. Do they make unrealistic assumptions?

Yes, the paper makes some assumptions about the client request type and size during doing the experiment, Capriccio also make assumptions about the memory control permission of the Server OS and WebProphet assumed that the HTTP Pipelining are disabled by default or without pipelining support for all the browser.

**Conclusion**