# Antiquity:
Secure Log For Wide-Area Distributed Storage

Administration Management Project

Michael Metral
Masters of Engineering – Computer Science
Course: CS 790 – Independent Research – Master's Project
Advisor: Dr. Hakim Weatherspoon
Cornell University

Summer 2007
Contact: mdm257@cornell.edu

**Abstract**

The goal of this document is to describe the concept, reasoning and construction of a management utility for Antiquity. Along with the actual management application is the discussion of the knowledge required in order to interact with design and actually program such systems. It is assumed that the reader is familiar with the *client-server model* amongst nodes as well as an acquaintance with *Remote Procedure Calls (RPC's)* and the *event-based programming* style (also referred to as *event-driven programming*). Note that the terms administrator/server and node/client are used interchangeably throughout this document.

**Antiquity Introduction**

Antiquity is a wide-area distributed storage system designed to provide a simple storage service and interface for applications like file-systems and back up. The design assumes that all servers eventually fail and attempts to maintain data despite those failures. Antiquity uses a secure log to maintain data integrity, replicates each log on multiple servers for increased durability, and uses quorum protocols to ensure consistency among replicas. Antiquity's design has been evaluated on global and local test beds. It was tested running on 400+ PlanetLab servers storing nearly 20,000 logs totaling more than 84 GB of data. Despite constant server churn, all logs remain durable.[1]

**Administration Management Introduction**

In its core, Antiquity maintains a hierarchy of several administrators each with their own set of children nodes. With the storage service Antiquity provides for file-systems it is important and practical to obtain and manage the statistics of each node and ultimately each administrator. Feedback of this kind allows administrators along with their peers (other administrators) to have knowledge of each other's systems and could ultimately lead to improvements to the locale of the data stored and maintained. Such statistics include but are not restricted to, disk usage, up and down time, deviation and average up and down time, availability, lifetime, and estimated round trip time for a statistic collection of a particular node.

However, no such management currently exists. With the requirement for such a tool, under the guidance of my advisor Dr. Hakim Weatherspoon, the topic for this Master's Project will revolve around the development and construction of this utility. The architecture of this application should not only serve to benefit the interested personnel from an administrative branch, but be designed efficiently and facilitated enough to extract, present and share the data provided by the collection of children nodes. Additionally, with the acquisition of these statistics the management utility could adapt itself to perform in an improved and timely fashion as to restrict any dragging nodes from decreasing the performance of the application and machines involved.

**Familiarization with Antiquity**

In order to construct an application for such an intricate system that uses

an uncommon coding style and other systems as a foundation, familiarization and practice with Antiquity is essential. With various techniques used in the creation of remote storage systems, there were necessary documents and articles that needed to be read in order to touch base with the terminology and design of Antiquity.

With a degree of sophistication inherent in the construction of Antiquity and systems-alike, research is ineffective without practice. As a means of interacting and working with this system from a clean slate, we decided on creating a simple chatting application amongst nodes connected to a central server, analogous to the design of Antiquity where nodes serving as storage hosts, communicate with an administrator.

**Event-Driven Programming & Staged Event-Driven Architecture (SEDA)**

Though an application such as a chat appears simple to construct, in this context it cannot be created successfully without first understanding the fundamentals. Antiquity is constructed in an event-driven programming style unlike the common flow-driven style widely used by the majority of programmers. Event-driven code differs from flow-driven in the fact that it is based on user actions rather than the flow determined by the programmer(s); thus, is required in any interactive application.

Event-driven programs need not require a command to carry out a specific task; rather, they operate by frequently processing any information they receive with a specific trigger for the incoming information in a looping fashion. They

consist of event queues, handlers and dispatchers all used to manage and process events asynchronously.

SEDA decomposes a complex, event-driven application into a set of stages connected by queues. This design avoids the high overhead associated with thread-based concurrency models, and decouples event and thread scheduling from application logic. By performing admission control on each event queue, the service can be well conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity.[2]

**Chat Application**

As a means of understanding the underlying SEDA structure along with Bamboo, an application written in the SEDA style that Antiquity uses as its basis, the idea of a simple chatting application among nodes was formed as a means of familiarizing beginners with such systems.

Given that a Bamboo application is composed of several stages, all of which communicate by passing events around, it is noted that this chat application will differ from the common chat implemented today. Using RPC's instead of existing protocols such as TCP/IP and UDP, in order to simulate a chat amongst several nodes a central node operating as a server will be triggered to multicast all incoming messages to all connected nodes. In other words, when a node wants to broadcast a message to a chat group they are currently connected to, they would request an event to send a text message from the central server and receive a response for the request. If successful, the server would in turn

request an event from all corresponding nodes of the group to forward the message, and would wait for a response from each node to recognize that they received the sender's text message.

**Programming Design and Requirements**

With the chat synopsis established, it is necessary to state the necessities to actually program any of these specific applications from a general level. All Bamboo applications require a declaration of all the stages to be implemented, their settings and an outline of the data types and definitions used. The configuration file is the housing file for the stages used in an application along with their settings; whereas an External Data Representation (XDR) file is used as a standard for the description and encoding of data types and definitions between different computer architectures and their communication. It is in an XDR file that each event is outlined by not only establishing the data types exchanged between machines, but also the arguments, results and an identifier triple (program number, version number, procedure number) that correspond to each.

Establishing the XDR file allows the capability to map an event to its data skeleton. It is with the aid of a dispatcher that events are designated to their appropriate destination by invoking event-handlers and storing all other unprocessed events in an event-queue. As events arrive, it is then possible to map the procedure number of the event to its corresponding method through these handlers. Eventually as the events are forwarded off to their modules, they

are carried out accordingly and a response to the event is created and dispatched back to the original requestor. As this pattern recursively continues, where one stage offers an event and another stage triggers the event, we can achieve a completely event-driven, single-threaded, interactive and asynchronous application.

**Stage Division**

We must indicate that each stage can be divided into three sections: the constructor, the initialization and the event-handler. The constructor, just like any traditional constructor is intended to pre-define an object's data members and validate the state of an object's class. The initialization process retrieves each stage's settings from the configuration file and establishes the inheritance and compatibility with Bamboo's Standard Stage. Lastly, the event-handler is the middle-man whose purpose is to direct an incoming event by procedure number to its appropriate method.

**Administration Management Application**

Establishing the prerequisites to interacting with Antiquity and Bamboo, we now focus on the administration management application. As previously stated, we would like to obtain and preserve statistical data for each of the children nodes under the control of the parent administrator. These collected statistics would present the administrator with a view of the node's current local state and in turn, any of the administrators peers. With the data, we can modify

the form of future state extraction from each node and allow the possibility for improvement to the conditions of the data stored on host nodes.

**Administration Management Application – Events & Stages**

In order to establish a clear definition for interaction between node and administrator we divide the management into three main events: registration of a node with the administrator, pinging a node from an administrative position to collect the node's state (their summary of statistics), and removal of a node from the administrator's authority.

Along with these events we require a set of *stages*, better known as a decomposition of an event-driven application, which communicate with one another to carry out a certain event. To conform to the underlying Bamboo system the primary stage we must extend is the Standard Stage. It is in this stage that many requisites, such as the structuring of event-handling to comply with future handlers for a specific event, are established.

Secondly, we must denote the separation of administrator and node by creating individual stages for each. These are the Statistics (Stats) Administrator and Stats Node respectively and each stage holds configurations relevant to their purpose.

Thirdly, we must distinguish the server and client functionalities with their own stages, these are known as the RPC Server Stage and RPC Client Stage respectively. It is necessary to observe that *administrator/node* differ from *server/client* in this document in the fact that *administrator/node* correspond to

labeling machines and *server/client* correspond to the actual role of hosting and requesting information. Visibly the RPC Server Stage and RPC Client Stage seem obvious enough to initially correlate to the administrator and node correspondingly; however, this is not enforced. It is the case in this application that both administrator and node must offer events to one another in order to achieve the ultimate goal of collecting statistical data. To further explain, a node must register or remove themselves with an administrator, so the administrator must be offering these capabilities to the node through the RPC Server Stage and the node must be requesting the event through the RPC Client Stage. However, to request the local state of a node, the node must be offering the capability to the administrator through the RPC Server Stage to collect their statistics and the administrator must request this event through the RPC Client Stage. In summary, both administrator and node implement the RPC Server Stage and RPC Client Stage.

Finally, the last stage required is the Storage Manager stage. This stage is responsible for maintaining a database of the local state relevant to each node, for the administrator.

**Administration Management Application – Event Process**

To further develop the three events previously listed: registration, pinging and removal of a node, we need to describe each events' structure and assure that they conform to Bamboo and SEDA policy with a request and callback function, or a handler and response.

In order to gain a general insight into event requirements we examine the terminology previously stated. A *request* is a demand for information or action sent by the client stage to the server stage. A *response* is a reply for a certain request sent by the server stage to the client stage. A *callback function* is a module that is assigned by to a specific request and is executed on the client stage end when the server stage responds to the client stage's request. Finally, a *handler* is a module that is assigned to deal with a specific request and is executed on the server stage end when the client stage makes the request to the server stage.

As a requirement, a node must register with an administrator to be under its authority and to be included in the statistical collection the administrator conducts. Conversely, a node must remove itself and be considered unregistered from the administrator's authority to no longer be included in the statistical collection. In regards to the administrator's role, it will perform numerous cycles where in each one, it probes all of the nodes registered with it on a fixed interval. It will continue to do this until it is triggered to terminate the operation.

Upon choosing to register with an administrator the node compiles a snapshot of its local state and sends it to the administrator along with the request to register. It would then wait for confirmation from the administrator that the registration was successful and if so, the administrator stores the node's initial local state. As to continually update the local state of a given node the administrator will acquire a node's information through the use of a ping. It is in response to this ping that the node once again rebuilds their information and

embeds it within the reply. After receiving the node's reply through the ping callback function, the administrator will use the embedded information to update the node's state.

However, if during a ping the administrator is unable to communicate with a node the administrator will continue the pinging process until they consider the node to have reached the pinging threshold. A *pinging threshold* is an integer set in the administrations configuration file limiting the amount of outstanding pings a node is able to commit during a given cycle. Upon reaching the threshold the administrator will consider the node down and mark this update in the nodes' state themselves since the node is incapable of broadcasting their failure. It is then up to the administrator to decide how to modify the frequency of the pings regarding that node. Note that it is in the administrator and other nodes best interest to not include the failed node for a given amount of time in order to avoid future delays.

The pinging process of a node is the crux of this management utility. It is the reasoning for a node to register or remove themselves from an administrator just as well as the incentive for assuring that permanent or transient failures of nodes are rationally handled. Given that the pinging of all nodes is done in a cycling fashion, we must distinguish the loop used in this utility from its counterpart. In this application we use a looping method known as an open loop. An *open loop* is the concept that an administrator will always issue a ping on a fixed interval regardless if a node was previously down. This differs from a *closed loop*, which is when the administrator will only ping a certain node if they've

previously received a response for an earlier ping. It is thus evident and beneficial in this application to use an open loop. The reasoning being that if a node happens to be subjected to permanent or transient failure then in a future cycle of the statistical collection it has an opportunity to update its local state with the administrator.

## Conclusion

Antiquity, a storage service and interface for file-systems and archived data required a management application capable of facilitating the process of obtaining statistical information for each branch of administrators. It is within this branch that several nodes are children to their parent administrator(s) and with the knowledge of their individual states, it is possible to use this data and its analysis to extend the integrity of the nodes and the file-system or data they are hosting. However, given the sophistication of a system like so, it is necessary and ideal to conduct research and practice relevant and beneficial to design and construct such a utility. With this management utility and a couple of additional features yet to be implemented, Antiquity could utilize this application to assemble an organized and managed set of branches and even have the opportunity to improve the nature of its decisions.

## References

[1] Antiquity Introduction. Hakim Weatherspoon. 2005-2007. http://antiquity.sourceforge.net

[2] SEDA Introduction. Matt Welsh. May 2006. http://www.eecs.harvard.edu/%7Emdw/proj/seda/