Distributed Systems

Assignment 2 - Documentation

James Bockman a1627392
James.Bockman@adelaide.edu.au
School of Computer Science,
The University of Adelaide

1 OVERVIEW

1.1 Composition

1.1.1 ***Aggregation Server***. Centeralised server responsible for the broadcasting and collating of feeds from the **Get/Put Clients** respectively.

1.1.2 ***Put Clients***. Responsible for providing the aggregation server with local content to be combined into the feed.

1.1.3 ***Get Clients***. Allow users to access the summary feed of content hosted by the aggregation server

1.2 ATOM XML

Provides a uniform standard for content to be transmitted between components of the system. Enables openness in the platform provided extensions utilize ATOM XML in their implementation for content transmission.

1.3 HTTP Requests

1.3.1 ***GET***. requests can be sent to the server to retrieve the latest version of the aggregated feed.

1.3.2 ***PUT***. requests can be sent to the server to request that the included ATOM XML content be hosted on the server and included in the feed disseminated to clients. On initial request the server will respond with 201 content created status and include an ETag for the content received in addition to a cookie. The ETag can be used by users hosting content to verify the XML being hosted is consistent with their content. Cookies are kept by users to include in further correspondence with the server. When a PUT request is received by the server with a cookie, the server will check if the cookie has content associated with it, then verify if the content sent matches what is on record. In this case no update is made, but the contents timeout window is refreshed. If the content sent with the PUT request is different to that being hosted, the current requests content will overwrite the previously held information for the client.

1.3.3 ***HEAD***. requests action depends on if the agent-user identifies as a get client or a put client. In this case of a get client the meta-data for the current version of the aggregated feed will be sent as a reply. Get clients can then use this data to ensure their current copy of the feed is update. Put clients can make successful HEAD requests after they have made a previous PUT request. In the HEAD request the put client should include the cookie assigned to them in the previous PUT transaction. This form of request will return the meta-information for that clients hosted content on the server and will refresh their contents timeout window.

Assignment 2 - Documentation, September 22, 2019, Adelaide, SA
2019.

2 AGGREGATION SERVER

2.1 Key Functionality

2.1.1 ***Update Feed***. Parses currently held content objects from Put Clients and creates an aggregated ATOM Feed object. This process is conducted on either GET or HEAD requests from Get Clients if a genuine PUT request has been processed, or if a feed has timed out since the last invocation of the aggregate() method.

2.1.2 ***Update Content***. Invoked implicitly on the receipt of a genuine PUT request. Operation preforms content validation, cookie assignment and updating the content records held. If a request does update the record a job is en-queued that will backup the client's information to file. **Send Feed** Invoked implicitly on receipt of **Get Client** GET request. Server responds with the most up to date version of the aggregated feed, calling aggregate() to reassemble the feed if any changes to hosted content have been made since the last call to aggregate() was made.

2.1.3 ***Send Meta-Date***. Invoked by HEAD request from both Put and Get clients. Agents identifying as Get clients receive meta information about the current aggregated feeds state, invoking aggregate() as required to ensure up to date information is provided. Request from Put clients with an assigned cookie will refresh the timeout window on the content associated with that cookie and respond with that contents meta-data.

2.1.4 ***Delete Expired Feeds***. At one second intervals a Garbageman Worker thread is instanced that will verify that clients last seen timestamps are no older than 15 seconds. Any records that have been present without contact from their associated client for 15 seconds will be removed from both the Content and Timestamp maps in addition to backed up meta and XML files. If a removal occurs aggregation will be required on the next Get Client HEAD or GET request received.

2.2 Request Management

Request Queue Requests for service made by clients of the server are received on the listening thread, which immediately hands the accepted socket to a Queue Worker. This worker parses the HTTP request for salient details and adds a Job object into a priority queue based on the Lamport Clock time provided in the parsed message. Job execution is handled by the main server thread which constantly polls the queue and instances the correct worker thread to handle the service.

2.2.1 ***Genuine Updates.*** To avoid **Put Clients** overwhelming the server with content updates a hashing is used to ensure a change in the content XML has actually occurred prior to requesting a write lock from the content map. **Probe Priority** A priority Probe service is integrated into the Queue worker which enables the server to respond to these immediately, bypassing any queued Jobs. A probe request can be sent by any client to establish that the server is live by sending a message consisting of a single byte. This request is answered by the server responding back with a single byte.

2.3 Shared Resource Management

Thread safety is ensured between workers on the aggregation server by using reader-writer locks on each of the shared resources used within the system. In this case this takes the form of three main content locks and a file writing lock. **Content Locks** Each of these three locks is implemented in the form of a reader-writer lock, enabling simultaneous read operation to occur. When a write request is made for a resource no further readers are permitted to enter the lock until all current readers have finished and the write operation completed. Threads are queued to access the lock using their associated requests Lamport Clock time. Write operations modify the object held in memory directly and reads return a copy of the shared resource as it was at the time of access. The following demands are present on each content object held:

Cookie to Content Map

- Put Worker (Read and Write)
- Get Worker (Read if aggregate required)
- Head Worker (Read)
- Checkpoint Worker (Read)
- Garbageman Worker (Write when content has expired)

Cookie to Timestamp Map

- Put Worker (Read and Write)
- Head Worker (Write)
- Get Worker (Read if aggregate required)
- Checkpoint Worker (Read)
- Garbageman Worker (Read and Write when content has expired)

Aggregated Feed

- Get Worker (Read and Write if aggregate required)
- Head Worker (Read and Write if aggregate required)

File Locks Threads responsible for maintaining back up content for hosted content require access to the file system. Two main actors in this area may run into conflict when a file is being updated by a Checkpoint Worker and simultaneously removed by a Garbageman Worker. To prevent this a simple single thread lock is used to prevent multiple workers modifying the file system at any one time. This solution is not salable and is marked for replacement with a lock that is managed separately for each client's records present on the file system. Allowing threads simultaneous access of different clients, but unique access to a specific one. As reading from the file system is only preformed when the sever is being initialised a more complex solution like that used to manage access to content is not required.

2.4 Fault Tolerance

2.4.1 Backup Files. Back up files are maintained by Checkpoint Workers ensuring hosted content and client meta-data is persistent between restarts. Two files are held for each client hosting content, an XML file containing the hosted content and a meta file containing both their logical and wall clock time stamps. These files are named with the cookie given to the client on their original PUT request. **Server Restart** On creation of the Aggregation servers process it checks local file content to verify if previously hosted records exist. These files are then read in sequentially to reconstruct the Cookie to Content and Cookie to Timestamps maps. As these are built the servers logical clock is also restored to the highest timestamp present within the meta files on record. Re-initialisation is finalised by calling aggregate to reconstruct the aggregated feed form the loaded content. Listening for clients and garbage collection is the resumed.

3 PUT CLIENTS

Responsible for providing the aggregation server with local content to be combined into the feed and maintaining a heartbeat indicating the content is accessible to users.

3.1 Key Functionality

3.1.1 ***Request Content Update.*** A HTTP PUT request is sent to the server with an attached copy of the clients content XML. If the client has been previously assigned a cookie by the server this should be included to avoid duplicate content being hosted.

3.1.2 ***Heartbeat.*** After receipt of a successful PUT request the client will make HEAD requests at regular intervals to ensure the server does not mark its content as expired.

3.2 Fault Tolerance

3.2.1 ***Client Restart.*** A meta file is created on receipt HTTP 201 content created that contains the client's cookie and ETag. These are loaded if available on re-initialisation of the process. After loading these values a HEAD request is sent to the server with the loaded cookie. The response from the server will either indicate that request was successful, returning an ETag for the hosted content that can be matched with the one locally held or that the service failed. If the latter is received the content present on the server has been removed due to timeout. The Put Client will then send a PUT request with its XML content attached to re-host its content on the server and resume its heartbeat.

3.2.2 ***Server Failure.*** In the case the server appears unresponsive to a clients request, indicated by a large wait for a response, an adaptive timeout is used. This will set an initial timeout time on the request before sending a probe to the server. If a positive response is received, indicating the server is able to respond, the timeout window is doubled and waiting on the initially reply is resumed. If no acknowledgement is received from z probe, it is retried three times at which point the server is assumed to be unavailable. Likewise if three timeout windows are failed with positive probe responses in between the server is assumed to be too busy to process the request. These results are communicated to the user indicate that

the service is currently unavailable and that they should try again later.

4 GET CLIENTS

4.1 Key Functionality

4.1.1 ***Request Updated Feed***. Sends a HTTP GET request to the Aggregation server which will provide the most update version of the feed as an attached XML string.

4.1.2 ***Check Feed***. Sends a HTTP HEAD request to the Aggregation server which will respond with the meta information for the most up to date version of the feed. Get clients can use this information to establish if a request for the updated feed is required.

4.1.3 ***Display Feed***. Unpacks the XML content into an object which is then used to display a formatted feed to the command line for users to read.

4.2 Fault Tolerance

4.2.1 ***Client Failure***. On restart the client behaves as it would if it had never previously existed. Establishing a connection to the aggregation server and requesting a copy of the XML feed.

4.2.2 ***Server Failure***. In the case the server appears unresponsive to a clients request, indicated by a large wait for a response, an adaptive timeout is used. This will set an initial timeout time on the request before sending a probe to the server. If a positive response is received, indicating the server is able to respond, the timeout window is doubled and waiting on the initially reply is resumed. If no acknowledgement is received from z probe, it is retried three times at which point the server is assumed to be unavailable. Likewise if three timeout windows are failed with positive probe responses in between the server is assumed to be too busy to process the request. These results are communicated to the user indicate that the service is currently unavailable and that they should try again later.

5 ENABLING CLASSES

5.1 LamportClock

Added to classes as a member object when inter-system synchronisation is required between other remote objects in the system. The classes constructor can either be called with or without an initial time. If no time is specified the clock is set to one.

5.1.1 ***void incrementTime(opt int time)***. If specified the optional argument time is compared to the current logical time of the clock. If it is larger it is used to set the value of the clock prior to incrementing the time. If no time is provided as an argument the local clock is incremented.

5.1.2 ***int getTime()***. Returns the current time of the logical clock.

5.1.3 ***void loadTime(int time)***. Used when restoring clients from meta files. Enables the setting of the logical time to a value without incrementing the clock. The method only allows the clock to be set higher than the current value.

5.2 XML Translator

Stand alone class that can be instanced to translate XML between required formats.

5.2.1 ***Input to XML:***. Parses a Put client's content file and creates an ATOM XML compliant message to be sent to the Aggregation server.

5.2.2 ***XML to XML:***. Parses ATOM XML files received by the Aggregation Server from Put clients and produces an ATOM XML compliant feed.

5.2.3 ***XML to Output:*** Parses an ATOM XML file sent from the Aggregation Server to an output format suitable for Get clients

5.3 HTTPHelper

5.3.1 ***String createHTTPResponse(String type, String content, String uuid, String senderName, int status, int logicalTime)***. Creates a string that can be sent via a socket that contains a HTTP compliant response assembled from the provided arguments. Accepts the type of request being responded to (Get, Put, Head), xml content, a cookie, name of the replying agent, status code and logical time. The type provides the contextual information required to handle how the content is used in the response. Explicitly this is whether it is used to construct meta-information from (Head and Put) or to be included as an attachment (Get).

5.3.2 ***String createHttpResponse(String senderName, int status, int logicalTime)*** . Wrapper for the previous method for when a reply has limited dependencies. Commonly used to construct replies for HTTP 400 and 500 status codes. Accepts the repliers agent name, status code and logical time. Returns a string containing the HTTP compliant response.

5.3.3 ***String createHTTPRequest(String type,String content, String senderName, String uuid, int logicalTime)***. Creates a string that can be sent via a socket that contains a HTTP compliant request assembled from the provided arguments. Accepts the request type (Get, Put, Head), xml content, requesting agent's name, agent's cookie and logical time.

5.3.4 ***HTTPResponse parseHTTPResponse(String)***. Parses a string received by a client from the server, creating a HTTPResponse object containing relevant information that is returned by the method. This object contains the logical time, ETag, HTTP status code, cookie, xml content and the name of the responding agent.

5.3.5 ***HTTPRequest parseHTTPRequest(String)***. Parses a string sent from a client creating a HTTPRequest object filled with relevant information that is returned by the method. The returned object contains the logical time, type of request, xml content and the requesting agent's class and cookie.

6 TESTING UTILITY

An optional constructor has been added to the **Aggregation Server**, **Get Client** and **Put Client** classes enabling the inclusion of a **Logger**. This in conjunction with the test flag being set to true will write key events into a log. This can be used to retrospectively analyse the behaviour of the systems key functions and messages sent and received by the actor. Each component is implemented as a runnable with a stop method that enables the creation of mock scenarios, starting and stopping components of the system as required. Currently the **TestUtil** class has eleven situations implemented that are each detailed below.

6.1 Designing Custom Tests

Instance a Logger for each of the components required for the scenario being tested and include them in their respective components constructor. While retaining a reference to each of the components instance a thread for each. Included methods can be called on the Thread or Component reference can be used to start and stop them in the required order. The logs and stored component meta files can also be reset as required.

6.1.1 ***Logger newLog(String agentName, String testName)***. Creates a new Logger that can be included when instancing components of the system. The agentName will specify the filename of the log associated with the Logger. testName provides the name of the folder that will be created by the Logger to store the logs in.

6.1.2 ***startServer(Thread server)***. Starts the passed thread and displays a message to the terminal indicating the action.

6.1.3 ***startPutClient(Thread putClient, String ref)***. Starts the passed thread and displays a message to the terminal indicating the action, with the included reference.

6.1.4 ***startGetClient(Thread getClient, String ref)***. Starts the passed thread and displays a message to the terminal indicating the action, with the included reference.

6.1.5 ***stopServer(AggregatedServer server)***. Calls the stop method on the passed AggregatedServer closing all sockets and ending any loops. Displays a message to the terminal indicating the action taken.

6.1.6 ***stopClient(PutClient putClient, String ref)***. Calls the stop method on the passed PutClient closing all sockets and ending any loops. Displays a message to the terminal indicating the action taken and including the reference passed in.

6.1.7 ***deleteMetaFiles()***. Clears any meta files stored by the AggregationServer and Put Clients from previous executions.

6.1.8 ***deleteLogs()***. Clears any testing logs present from previous tests.

6.1.9 ***sleep(int seconds)***. Sleeps for the number of seconds specified

6.2 Get Request

Server is initialised and waiting to service requests. A Get client is started and requests the feed from the server. The server then responds with the empty feed which is then displayed to the user.

6.3 Put Request without content

Server is initialised and waiting to service requests. A Put client is started and requests to host its content on the server, without including any content. It is expected that the server replies with a HTTP 204 no content response.

6.4 Put Request with content

Server is initialised and waiting to service requests. A Put client is started and requests to host its content on the server. It is expected that the server replies with a HTTP 201 content created response, including a cookie and Etag which is stored locally in a meta file. The Put client should then start a heartbeat to the server making regular HTTP Head requests with its cookie.

6.5 Put sequential Get

Server is initialised and waiting to service requests. A Put client is started and requests to host its content on the server. It is expected that the server replies with a HTTP 201 content created response, including a cookie and Etag which is stored locally in a meta file. The Put client should then start a heartbeat to the server making regular HTTP Head requests with its cookie. A Get client is then started that will request the feed from the server and display it to the user.

6.6 Multiple Put sequential Get

Server is initialised and waiting to service requests. Several Put clients are started, hosting their content on the server. A Get client is then started that requests the aggregated feed from the server.

6.7 Put multiple sequential Gets

Server is initialised and waiting to service requests. A Put client is started, hosting its content on the server. Several Get clients are started that request the aggregated feed from the server.

6.8 Multiple Put multiple Gets

Server is initialised and waiting to service requests. Several Put clients are started, hosting their content on the server. Several Get clients are started that request the aggregated feed from the server.

6.9 Interleaved Puts and Gets

Server is initialised and waiting to service requests. Put and Get clients are started one after the other, hosting their content on the server and requesting feeds. Expected behaviour is that each sequential Get client's log shows a feed with more entries.

6.10 Feed Expiry due to non-contact

Server is initialised and waiting to service requests. A Put client is started and requests to host its content on the server. A Get client is started to verify the initial request had been added to the feed. The Put client is then stopped and a wait time ensues. After this a Get client is started that should retrieve a feed that now does not contain the previously observed content.

6.11 Server Restart during operation

Server is initialised and waiting to service requests. A Put client is started and requests to host its content on the server. A Get client is started to verify the initial request had been added to the feed. The server is then stopped and a wait time ensues. After this a new server process is started, loading the previous records it has stored on file. A Get client is started that should retrieve a feed the same as the previously obtained one.

6.12 Put Client Restart during operation

Server is initialised and waiting to service requests. A Put client is started and requests to host its content on the server. A Get client is started to verify the initial request had been added to the feed. The Put client is then stopped and a wait time ensues. After this a new put client process is started, loading the previous records it has stored on file. A Get client is started that should retrieve a feed the same as the previously obtained one.

6.13 Put Client Failure during operation

Server is initialised and waiting to service requests. A Put client is started and requests to host its content on the server. A Get client is started to verify the initial request had been added to the feed. The Put client is then stopped and a wait time ensues. A Get client is started that should retrieve a feed the same as the previously obtained one. After another wait time a Get client again retrieves the feed, which this time is empty as the Put client's content has expired.