U.PORTO

FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

Distributed backup service

Masters in Informatics and Computing Engineering
Distributed Systems
Class 4 – Group 12
27th of May, 2018

Francisco Teixeira Lopes ei11056@fe.up.pt
Bruno Miguel Faustino Moreno up201504781@fe.up.pt

Table of Contents

Introduction

The developed application has for its main feature, the backup of files to a local P2P system. The files can be backed up and later restored or deleted. Each Peer manages its own storage area with limited disk space, whenever disk space runs out, the Peers are responsible for deleting what they find to be less relevant. This might trigger a rebalance of the chunks in the system so that chunks don't get permanently lost.

When compared to its previous iteration, the current system has a few improvements, chiefly the implementation of the DELETE and RESTORE enhancements, as well as security considerations and a bit more scalability. Some degree of fault tolerance has also been included.

This report will start by specifying the architecture of the application, followed by implementation details and finally some of the aspects behind the advanced features developed.

Architecture

Being a P2P application, the main component is the Peer class, this component is responsible for keeping a reference to every other component in the system.

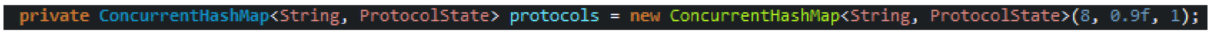
The ServiceChannel family of classes is responsible for the main communication aspect of the application, it receives UDP multicast messages and dispatches them to a handler for processing.

The SystemHandler and TimeoutHandler classes are responsible for processing the dispatched messages from the service channels.

Seeing as each Peer keeps a database of what they think is in the system, the SystemDatabase class facilitates this.

For security considerations, each Peer has its own SecurityHandler, responsible for handling MAC generation and AES-128 encryption/decryption.

The system runs mainly on two big blocks, the channels receiving messages and respective message handlers, and the protocol threads running a protocol instance. To address the communication between these two components, a ProtocolState class keeps the state of each running protocol. These states are stored in a hash map with various kinds of textual keys to identify the protocol, the most common one being "PeerID + FileID + ProtocolType", but several others exist when disambiguation is needed.



```java
private ConcurrentHashMap<String, ProtocolState> protocols = new ConcurrentHashMap<String, ProtocolState>(8, 0.9f, 1);
```

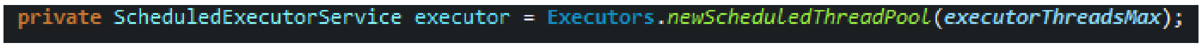
Figure 1 - the active protocols of a Peer (Peer.java line 77)

Implementation

Concurrency Design

V1.0 Design

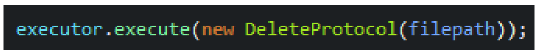
The Peer maintains a ScheduledExecutorService for running protocols and timeout threads, as pictured.



```java
private ScheduledExecutorService executor = Executors.newScheduledThreadPool(executorThreadsMax);
```

Figure 2 - the Peer's scheduled executor service field (Peer.java line 78)

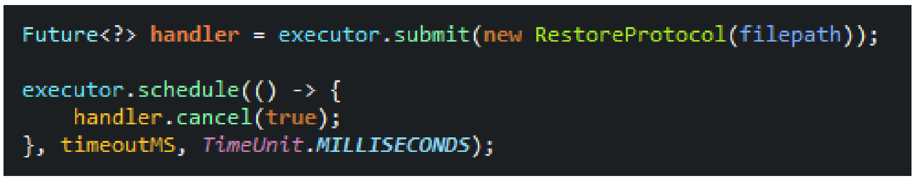
Whenever the test client invokes a system protocol, an executor thread is created to handle the request.



```java
executor.execute(new DeleteProtocol(filepath));
```

Figure 3 - test client requests threading implementation (Peer.java line 286)

If the request happens to be a RESTORE protocol then the invocation uses a scheduled timeout, which allows the RESTORE to run for a set amount of time before being considered that the file cannot be restored. The timeout is proportional to the total amount of chunks to be restored.

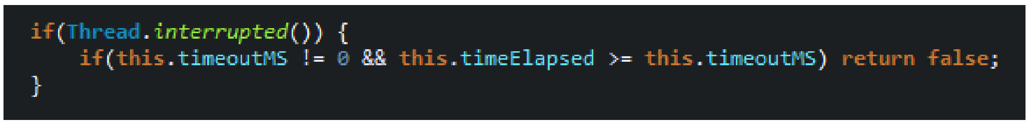


```java
Future<?> handler = executor.submit(new RestoreProtocol(filepath));

executor.schedule(() -> {
    handler.cancel(true);
}, timeoutMS, TimeUnit.MILLISECONDS);
```

Figure 4 - RESTORE protocol scheduled thread (Peer.java line 276)

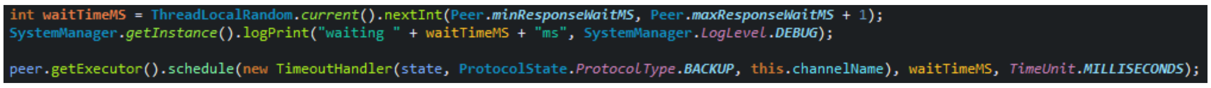
The thread running a RESTORE protocol instance gets interrupted by the “handler.cancel(true)” call and stops whenever it next polls the interrupted flag.



```java
if(Thread.interrupted()) {
    if(this.timeoutMS != 0 && this.timeElapsed >= this.timeoutMS) return false;
}
```

Figure 5 - checking interrupted flag (RestoreProtocol.java line 264)

As said before, the timeout threads also use this executor service to run the random delay before sending response messages. This is done by scheduling a thread to run after a delay has elapsed.



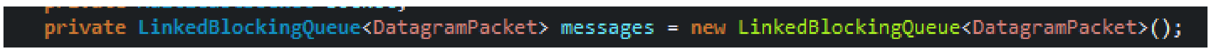
```java
int waitTimeMS = ThreadLocalRandom.current().nextInt(Peer.minResponseWaitMS, Peer.maxResponseWaitMS + 1);
SystemManager.getInstance().logPrint("waiting " + waitTimeMS + "ms", SystemManager.LogLevel.DEBUG);

peer.getExecutor().schedule(new TimeoutHandler(state, ProtocolState.ProtocolType.BACKUP, this.channelName), waitTimeMS, TimeUnit.MILLISECONDS);
```

Figure 6 - timeout handler implementation (SystemHandler.java line 133)

V2.0 Additions

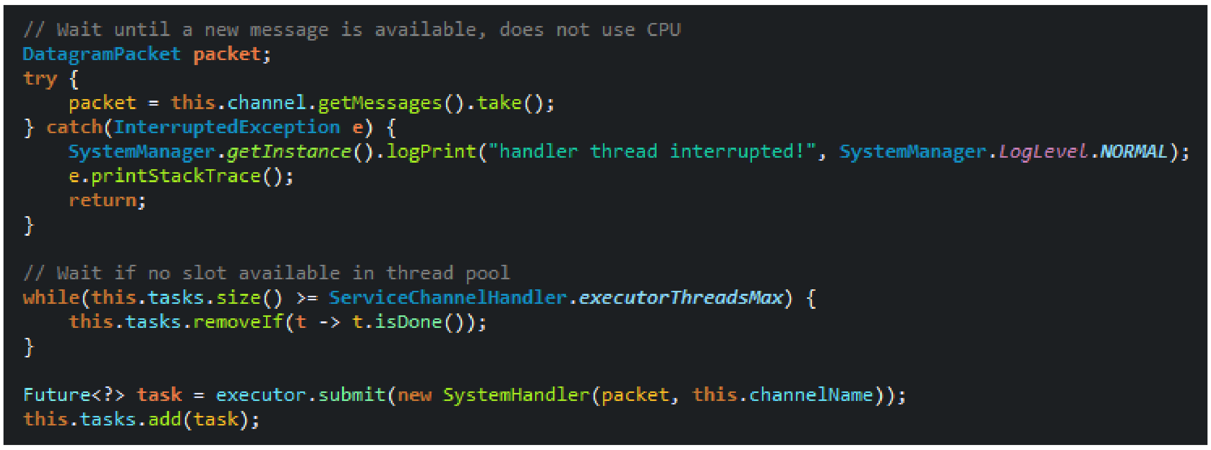
Each service channel inserts the received message in a queue for later handling by a constantly running handler thread.



```java
private LinkedBlockingQueue<DatagramPacket> messages = new LinkedBlockingQueue<DatagramPacket>();
```

Figure 7 - message queue (ServiceChannel.java line 16)

The handler thread submits new tasks to an ExecutorService to process queue messages and waits if no threads are available due to congestion. This makes message dropping a non-issue and the worst-case scenario is a timeout of the message. Which due to resending and Peer redundancy isn't a real issue either.



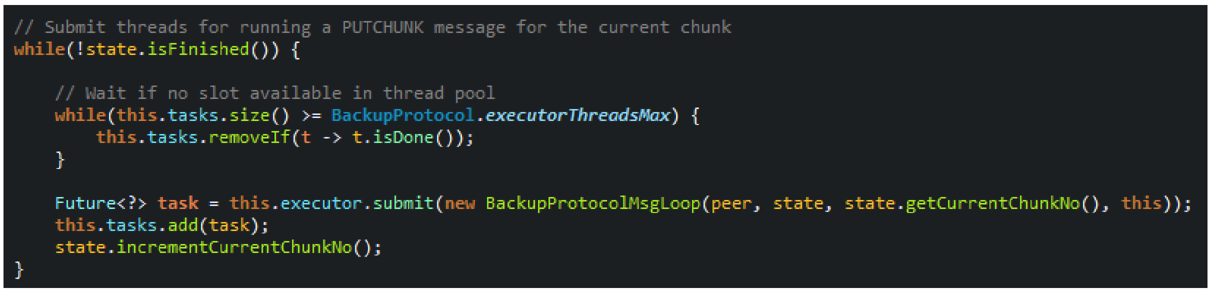
```java
// Wait until a new message is available, does not use CPU
DatagramPacket packet;
try {
    packet = this.channel.getMessages().take();
} catch(InterruptedException e) {
    SystemManager.getInstance().logPrint("handler thread interrupted!", SystemManager.LogLevel.NORMAL);
    e.printStackTrace();
    return;
}

// Wait if no slot available in thread pool
while(this.tasks.size() >= ServiceChannelHandler.executorThreadsMax) {
    this.tasks.removeIf(t -> t.isDone());
}

Future<?> task = executor.submit(new SystemHandler(packet, this.channelName));
this.tasks.add(task);
```

Figure 8 – the message dispatcher (ServiceChannelHandler.java line 32)

Lastly, the threading of the BACKUP protocol has also been improved, it now runs a set amount of PUTCHUNK messages at the same time for quicker backups.



```java
// Submit threads for running a PUTCHUNK message for the current chunk
while(!state.isFinished()) {

    // Wait if no slot available in thread pool
    while(this.tasks.size() >= BackupProtocol.executorThreadsMax) {
        this.tasks.removeIf(t -> t.isDone());
    }

    Future<?> task = this.executor.submit(new BackupProtocolMsgLoop(peer, state, state.getCurrentChunkNo(), this));
    this.tasks.add(task);
    state.incrementCurrentChunkNo();
}
```

Figure 9 - the BACKUP protocol concurrency (BackupProtocol.java line 57)

DELETE enhancement

To make sure chunks were deleted, a DELETED confirmation message was added. If the expected Peer IDs do not respond, the Peer running the DELETE protocol stores the missing Peer IDs in its database for later deletion.

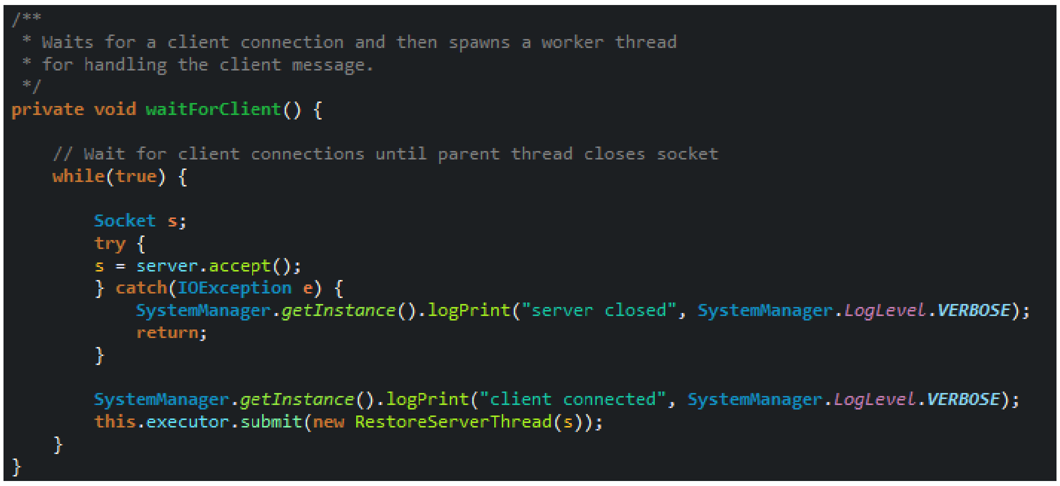
Whenever a Peer starts it sends a STARTED message which triggers any pending deletions stored on the other Peers' databases.

The problem could be solved by the backup initiator Peer having to renew a lease on the chunks it currently has backed up, but this avenue was abandoned due to time constraints.

RESTORE enhancement

For implementing the RESTORE protocol enhancement, the initiator Peer starts a TCP server which the other Peers, who own the chunk requested, connect to. The IP/port of the server is sent along with the GETCHUNK message on a second header line for interoperability. The Peer running the TCP server submits a new thread for every client connected to store the chunks non-sequentially. The number of concurrent GETCHUNKs is determined by a set value which can be changed to raise the concurrency level of the enhancement.

Each Peer reserves up to 10 ports for the RESTORE enhancement, allowing every Peer to run up to 10 RESTOREs at the same time. Although they can't RESTORE the same file since the CHUNK message lacks a destination parameter which makes Peers stop trying to connect if they find a CHUNK message for the same chunk they were going to provide.



```java
/**
 * Waits for a client connection and then spawns a worker thread
 * for handling the client message.
 */
private void waitForClient() {

    // Wait for client connections until parent thread closes socket
    while(true) {

        Socket s;
        try {
        s = server.accept();
        } catch(IOException e) {
            SystemManager.getInstance().logPrint("server closed", SystemManager.LogLevel.VERBOSE);
            return;
        }

        SystemManager.getInstance().logPrint("client connected", SystemManager.LogLevel.VERBOSE);
        this.executor.submit(new RestoreServerThread(s));
    }
}
```

Figure 10 - TCP server client waiting (RestoreServer.java line 30)

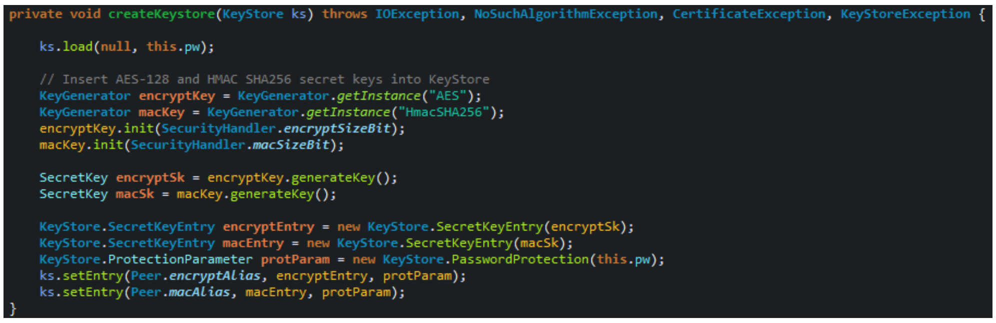
The server submits a new RestoreServerThread which then waits for the client to send an Object through an ObjectInputStream which represents the chunk data.

Relevant issues

Security

The application is meant to run on a company's intranet, as such, each Peer only needs a password to run any security measures. This password would be provided to select personnel.

The password unlocks a Java keystore common to all Peers, which contains a secret key for generating a SHA256 MAC and another secret key to encrypt/decrypt with AES-128.



```java
private void createKeystore(KeyStore ks) throws IOException, NoSuchAlgorithmException, CertificateException, KeyStoreException {

    ks.load(null, this.pw);

    // Insert AES-128 and HMAC SHA256 secret keys into KeyStore
    KeyGenerator encryptKey = KeyGenerator.getInstance("AES");
    KeyGenerator macKey = KeyGenerator.getInstance("HmacSHA256");
    encryptKey.init(SecurityHandler.encryptSizeBit);
    macKey.init(SecurityHandler.macSizeBit);

    SecretKey encryptSk = encryptKey.generateKey();
    SecretKey macSk = macKey.generateKey();

    KeyStore.SecretKeyEntry encryptEntry = new KeyStore.SecretKeyEntry(encryptSk);
    KeyStore.SecretKeyEntry macEntry = new KeyStore.SecretKeyEntry(macSk);
    KeyStore.ProtectionParameter protParam = new KeyStore.PasswordProtection(this.pw);
    ks.setEntry(Peer.encryptAlias, encryptEntry, protParam);
    ks.setEntry(Peer.macAlias, macEntry, protParam);
}
```

Figure 11 - secret key generation and insertion (KeystoreManager.java line 76)

Every service message is appended the computed MAC which is then used on the receiver to validate the message by computing the MAC again and seeing if it matches the appended MAC. The appended MAC is, of course, not used to compute the new MAC. This deals with the problems of message authentication and message tampering.

For message encryption/decryption AES-128 is used on the chunk data making it no longer possible to join the chunks and checking their contents without going through the backup service and the required security clearances.

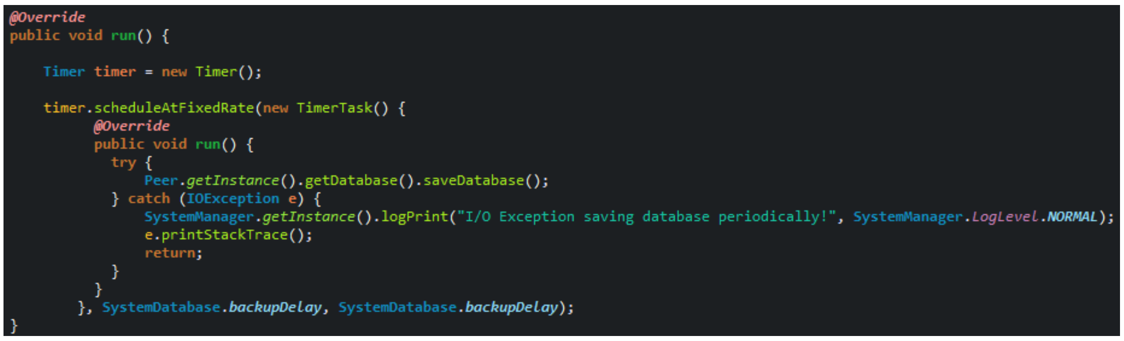
HMAC SHA256 can be seen in SecurityHandler.java line 29.

AES-128 can be seen in SecurityHandler.java line 74 and 127.

All algorithms and keystore types used come standard with the Java distribution to mitigate problems between different distributions.

Fault Tolerance

The database of each Peer gets periodically saved to disk so it doesn't get lost when a Peer fails. This saved file is loaded whenever the Peer starts.

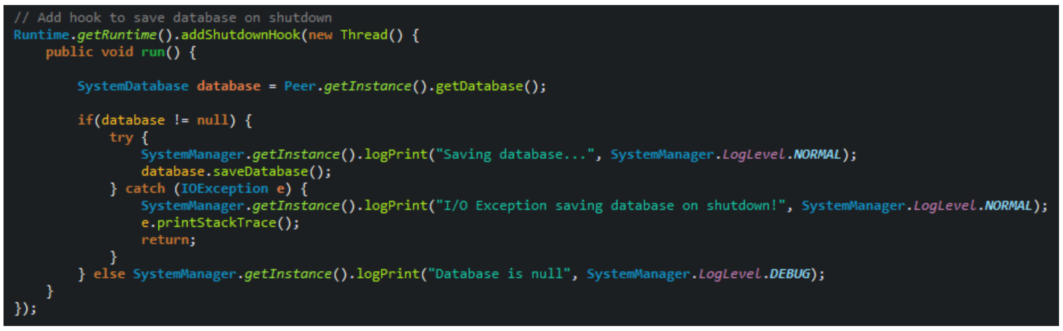


```java
@Override
public void run() {

    Timer timer = new Timer();

    timer.scheduleAtFixedRate(new TimerTask() {
          @Override
          public void run() {
            try {
                Peer.getInstance().getDatabase().saveDatabase();
            } catch (IOException e) {
                SystemManager.getInstance().logPrint("I/O Exception saving database periodically!", SystemManager.LogLevel.NORMAL);
                e.printStackTrace();
                return;
            }
          }
        }, SystemDatabase.backupDelay, SystemDatabase.backupDelay);
}
```

Figure 12 – database saving thread (SystemDatabase.java line 305)



```java
// Add hook to save database on shutdown
Runtime.getRuntime().addShutdownHook(new Thread() {
    public void run() {

        SystemDatabase database = Peer.getInstance().getDatabase();

        if(database != null) {
            try {
                SystemManager.getInstance().logPrint("Saving database...", SystemManager.LogLevel.NORMAL);
                database.saveDatabase();
            } catch (IOException e) {
                SystemManager.getInstance().logPrint("I/O Exception saving database on shutdown!", SystemManager.LogLevel.NORMAL);
                e.printStackTrace();
                return;
            }
        } else SystemManager.getInstance().logPrint("Database is null", SystemManager.LogLevel.DEBUG);
    }
});
```

Figure 13 - shutdown hook to save database even on crashes (Peer.java line 127)

Furthermore, the restore and delete protocols no longer require the original files to generate the service file ID. The peer first checks its own database to see if the file ID is stored there, if not, it sends a RETRIEVE message with the filepath/filename passed as command line argument to see if any other Peer knows the file ID.

The biggest improvement to fault tolerance would be the backing up of the databases and the keystore, since only one copy exists per Peer a I/O failure would make the database and keystore inaccessible. If the backup protocol and restore protocol were used, these files could be backed up in the service as special chunks, which would use the SHA256 of the filename as file ID. This would be more than enough since every Peer uses a unique filename for the database.

Conclusion

The second iteration of the distributed backup service features a lot of improvements over the first iteration, mainly to do with missing enhancements, security considerations, additional scalability and some fault tolerance considerations. Despite this, there are some aspects which could be improved upon, some have been referenced in previous sections but will be referenced here again.

The concurrency model of the application has almost no unneeded waiting periods, but the application I/O is still a blocking call which could be lifted by using the Java NIO functions. Regarding the DELETE enhancement, leases could've been used as mentioned. And finally, for a decent boost in the fault tolerance of the system, the Peer's databases and system keystore could be backed up in the backup system to use as fall back in case of I/O failure.

Overall the group considers the project a success and feels that the course's goals have been successfully ingrained over the course of the semester.