UNIVERSITEIT ANTWERPEN

Academiejaar 2016-2017

Faculteit Toegepaste Ingenieurswetenschappen

5-Gedistribueerde Systemen Practica

Lennert Van Hasselt, Liam Oorts, Thijs Anthonis, Jonas Vercauteren

Bachelor of Science in de industriële wetenschappen: elektronica-ICT

5-Gedistribueerde Systemen

Universiteit Antwerpen

Inhoudsopgave

Inleiding

In het hedendaagse leven gebruiken we steeds meer en meer applicaties zoals dropbox, onedrive en google drive om documenten in op te slaan. De vraag is echter hoe zit de werking hier achter. Voor het practica van 5-gedistribueerde systemen proberen we een simpele vorm van zulk systeem te ontwerpen genaamd ‘System Y’. Het hele project is geschreven in java en moest voldoen aan bepaalde vooropgestelde eisen. Deze eisen zullen kort besproken en toegelicht worden voor onze uitwerking.

Naming Server

Voor dit deel van het practica werd de basis voor de komende sessies uitgerold. We bespreken de uitwerking aan de hand van de gemaakte Classes.

MainServer

Beginnende zal de server opgestart worden in de class **MainServer.java**. Het eerste wat er hier zal gebeuren is het aanmaken van een **ClientInfo**, deze klasse wordt voornamelijk gebruikt voor de lijst van de verschillende nodes met bijhorende ip-adres bij te werken. Ook zit in deze klasse de hash-functie, deze neemt een hash van de naam van de node, dit getal deelt het dan door 2^15 en de rest is de hash. Hierdoor ligt deze waarde tussen 0 en 2^15.
In ClientInfo zal er een list van het type **ListNodes** aangemaakt worden. Hierin worden alle nodes met bijhorende ip-adressen dan opgeslagen. Deze informatie, in de vorm van een TreeMap, wordt dan opgeslagen op de harde schijf door serialization. Voor elke bewerking/opzoeking van informatie openen we dit object dan terug en slagen we deze weer op. De lijst is dan veilig opgeslagen, ook al crashed de server. Bij opstarten van de server checkt het systeem dan of er een file is die deze kan gebruiken, zoniet creëert hij zelf een file.
We willen ook dat als 2 nodes toevallig **dezelfde hash-waarde** hebben, dat het systeem deze wel apart opslaat. We gaan dan bij een toevoeging van een nieuwe node, eerst checken of deze hashwaarde al in de map aanwezig is, zo ja verhogen we de hash met 1 en checken we terug.
Om een **file op te zoeken** in het systeem gaan we deze eerst hashen, dit geeft de server dan de locatie waar deze file zou moeten zitten. We checken hiervoor eerst of de hash niet groter is dan de laatste nodenummer in het systeem, als dit het geval is geeft de mainserver als locatie de eerste node terug. De file moet namelijk aanwezig zijn op de eerstvolgende node in het systeem.

MainNode

Naast de MainServer zal **MainNode.java** de volledige functionaliteit bevatten van de gebruikers van het systeem. Beginnende met het toetreden ervan. De gebruiker kiest een naam en een nieuwe Class node zal aangemaakt worden. Door middel van een Multicast (**MulticastSender**) naar de server wordt de info (zijn naam en ip) verzonden en zal zijn naam gehashed worden. De verkregen gehashte waarde zal dan toegevoegd worden aan zijn list. en teruggezonden worden naar de MainNode. De eerst ontmoeting tussen server en gebruiker is zo gebeurd waarna ze elkaar kunnen gaan bereiken via IP. Voor de werking van de node verder uitgelegd zal worden bekijken we eerst multicast en RMI nader.

Multicast

Multicast wordt gebruikt om alle gebruikers aanwezig op het netwerk in te lichten over de aanwezigheid van een nieuwe node. De **MulticastSender** zal in dat geval een MulticastSocket aanmaken welke gekoppeld is aan een universeel adres (225.1.1.1) dat toegang heeft tot het gehele netwerk. De receiver, **MulticastReceiverServer** zal in slaapstand staan tot hij iets ontvangt. De verzonden multicast pakketten zijn van het type DatagramPacket en worden via de UDP poort 8888 verzonden. Wanneer de server de multicast beantwoord zal de node met zijn **MulticastReceiver** de gehashte waarde van de specifieke node terugkrijgen voor verder gebruik.

RMI

Om nu te werken met een gerichte communicatie tussen de gebruiker en de server via IP, zal Remote Method Invocation gebruikt worden. Om Remote Method Invocation of verkort geschreven RMI te gebruiken zal deze eerst geïnitialiseerd moeten worden. We definiëren Naming.bind() met als parameters de bindlocatie voor de RMI en het IP naar wie hij moet verzenden. Verder specificeert hij ook dat er op poort 1099 ontvangen en verzonden zal

worden. Om nu communicatie via RMI te verkrijgen met de server vereist dit bij de MainServer een initialisatie analoog opgebouwd aan deze in MainNode.

Om bepaalde functies op te kunnen roepen van de server moet er een Interface zijn die geïmplementeerd is bij de ClientInfo. Deze ClientInfo zal net zoals Node extended UnicastRemoteObject zijn. De interface is dan weer extended door Remote. De functies die dus opgeroepen worden in ClientInfo staan ook in de interface. Het oproepen hiervan loopt steeds hetzelfde:

```
String name = "//"+IP van de ontvanger+"/cliNode";
ClientInterface cf = (ClientInterface) Naming.lookup(name);
node.setClientInterface(cf);
```

Node

Nu Multicast en RMI verklaard zijn kan de functionaliteit in node aangehaald worden. Voor dit onderdeel is het beperkt tot het toevoegen van een node, het opzoeken van een bestand met een bepaalde bestandsnaam en het verwijderen van een node.

1. Toevoegen van een Node zal niet via RMI gebeuren maar gebeurd direct na het ontvangen van de Multicast bij de server.
2. Het opzoeken van een bestand gaat wel via de node, De gebruiker geeft een filenaam in met de juiste extensie en zal via de RMI de functie *cf.searchFile(String search)* oproepen. Als resultaat krijgt men hierbij een hash terug en het IP. Wanneer de gezochte file gevonden is pingen we dit IP om te zien of deze bereikbaar is voor ons. Om dit uit te kunnen voeren werd *Runtime.getRuntime().exec("ping "+IP)* gebruikt waarbij IP stond voor het IP van de gebruiker waar de file staat op gemapt.
3. Het verwijderen van een node is voor de gebruiker niet meer dan een RMI oproep doen naar de functie *cf.deleteNode(ownNode)* hierbij is ownNode de hashwaarde van de huidige node.

Node Life Cycle

Opstarten (Bootstrap + Discovery)

Bij het toevoegen van een nieuwe node, zal de node zelf een multicast pakket verzenden. Dit pakket zal de naam van de server bevatten. De server en al de aanwezige nodes zullen dit pakket ontvangen en verwerken.

De server zal bij het ontvangen van het pakket de functie *setNode(String clientName, InetAddress IP)* aanroepen. Parameters van deze functie kan de server achterhalen aan de hand van de ontvangen multicast.
Deze functie zal de clientName hashen (met *hashing(String name)*). De bekomen hash zal de server vervolgens vergelijken met alle aanwezige nodes om te controleren of deze hash al in gebruik is. Als dit het geval is zal de hash met 1 vermeerderd worden. Deze hash zal dan opgeslagen worden in **ClientInfo**.
Vervolgens zal de server de nieuwe node contacteren door de methode *setNameServer(String ip, int ownNode, int totalNodes)* via RMI aan te roepen. TIjdens deze methode zal de nieuwe node het ip van de server, zijn eigen node nummer, zijn eigen ip en het totaal aantal nodes opslaan. In het geval dat er geen andere nodes aanwezig zijn, zal deze methode de previous en next node veranderen naar de node zelf.

De andere al aanwezige nodes zullen bij het ontvangen de methode *hashing(String name, InetAddress IPraw)* aanroepen. Welke zal bepalen of de nieuwe node de next of previous node wordt.

Om dit gestructureerd te controleren, hebben wij drie situaties gedefinieerd.
In de eerste situatie was de ontvangende node alleen in het systeem. In dit geval is het heel gemakkelijk: de nieuwe node wordt de previous en next.
Bij de tweede situatie zitten er maar twee nodes in het systeem. In dit geval zijn er een hoop verschillende soorten gevallen van hoe het systeem eruit gaat zien. In de code zijn al de gevallen uitgelegd en visueel voorgesteld.
Hieronder staat een beschrijven van elke situatie:

- Prev/next < own
  - ----------New----------Prev/Next----------Own----------

    In dit geval is de hash van de nieuwe node kleiner dan de previous/next node. Hier zal de next node veranderd moeten worden.

  - --------Prev/Next----------New----------Own----------

    In dit geval ligt de hash van de nieuwe node tussen de eigen node en de prev/next node. Hier zal dus de previous node veranderd moeten worden.

  - ---- Prev/Next=New----------Own----------

    In dit geval is de hash van de nieuwe node gelijk aan de prev/next node. De hash van de nieuwe node zal met 1 worden vermeerderd. Ook de previous node zal veranderd worden.

  - --------Prev/Next----------Own----------New----------

    In dit geval is de hash van de nieuwe node groter dan de eigen node. Hier zal de next node aangepast worden.

  - --------Prev/Next----------Own=New----------

    In dit geval is de hash van de nieuwe node gelijk aan de eigen node. De hash van de nieuwe node zal met 1 worden vermeerderd. Ook de next node zal veranderd worden.

- Prev/next > own
  - ----------New----------Own----------Prev/Next----------

    In dit geval is de hash van de nieuwe node kleiner dan de eigen node. Hier zal de previous node veranderd moeten worden.

  - ----------Own----------New----------Prev/Next----------

    In dit geval ligt de hash van de nieuwe node tussen de eigen node en de prev/next node. Hier zal dus de nextnode veranderd moeten worden.

  - --------Own=New----------Prev/Next----------

    In dit geval is de hash van de nieuwe node gelijk aan de eigen node. De hash van de nieuwe node zal met 1 worden vermeerderd. Ook de next node zal veranderd worden.

- ----------Own----------Prev/Next----------New----------

  In dit geval is de hash van de nieuwe node groter dan de prev/next node. Hier zal de previous node aangepast worden.

- ----------Own----------Prev/Next=New------

  In dit geval is de hash van de nieuwe node gelijk aan de prev/next node. De hash van de nieuwe node zal met 1 worden vermeerderd. Ook de previous node zal veranderd worden.

In de laatste situatie bevinden er zich drie of meer nodes in het systeem. Ook hier zijn er veschillende gevallen. Hieronder worden deze uitgelegd:

- own < next < previous
  - ----------Own----------Next----------Prev----------New----
    of
    ----------New----------Own----------Next----------Prev----

    In deze beide gevallen bevindt de nieuwe node zich tussen de eigen node en de previous node. Hier zal de previous node veranderen.

  - ----------Own----------New----------Next----------Prev----

    In dit geval bevindt de hash van de nieuwe node zich tussen de eigen node en de next node. Hier zal de next node aangepast moeten worden.

- previous < own < next
  - ----------Prev----------New----------Own----------Next----

    In dit geval bevindt de nieuwe node zich tussen de eigen node en de previous node. Hier zal de previous node veranderen.

  - ----------Prev----------Own----------New----------Next----

    In dit geval bevindt de hash van de nieuwe node zich tussen de eigen node en de next node. Hier zal de next node aangepast moeten worden.

- next < previous < own
  - ----------Next----------Prev----------Own----------New----
    of
    ----------New----------Next----------Prev----------Own----

    In deze beide gevallen bevindt de nieuwe node zich tussen de eigen node en de next node. Hier zal de next node veranderen.

  - ----------Next----------Prev----------New----------Own----

    In dit geval bevindt de nieuwe node zich tussen de eigen node en de previous node. Hier zal de previous node veranderen.

Het is belangrijk om te onthouden dat telkens wanneer de ontvangende node zijn next node moet aanpassen, de ontvangende node via RMI de nieuwe node zal contacteren om de

methode *changePrevNext(int nextNode, int previousNode, String nextIP, String previousIP)* uit te voeren. Hierdoor zal de nieuwe node waar hij zich in het systeem bevindt.

Shutdown

Wanneer een node zich uit het systeem wilt verwijderen, dan zal er vanalles moeten gebeuren in het systeem. De methode *deleteNode()* zal aangesproken worden. Hier zal de node als eerste al de bestanden die op deze node gerepliceerd zijn doorgestuurd worden naar zijn previous node. Dit zal enkel gebeuren als de previous en next node niet dezelfde zijn, want in dat geval zal er maar één node in het systeem overblijven.
Om de bestanden naar de previous node te sturen bestaat de methode *sendReplicatedFilesToPrevious()*. Deze methode zal bij replicatie verder uitgelegd worden.
Nadat al de bestanden verstuurd zijn, zal de node zijn buren contacteren via RMI. Dit zal enkel gebeuren als de node zelf niet de enige node in het systeem is. Bij zijn previous node zal hij de methode *setNextNode(int nextNode, String nextIP)* aanroepen, met als meegegeven parameters de gegevens van zijn nextnode. Op een gelijkaardige manier zal op de next node aangesproken worden om de methode *setPreviousNode(int previousNode, String previousIP)* uit te voeren, de meegegeven parameters zijn de gegevens van zijn previous node.
Als dit gebeurt is, kan de node de server contacteren door de methode *deleteNode(int aNode)* via RMI aan te roepen. De server zal de node dan uit zijn clientInfo verwijderen.
Pas op dit moment is de node volledig uit het systeem verdwenen.

Failure

Een failure van een node zal pas ontdekt worden als een andere node deze gefailde node wilt contacteren. Als dit niet lukt zal het foutlopen van het opzetten van een verbinding tussen deze twee nodes, gecatched worden. In deze catch zal de methode *updateNetwork(int node)* uitgevoerd worden.
In deze methode zal als eerste aan de server gevraagd worden met *getPreviousNext(int node)* wie de buren van de gefailde node zijn. Deze methode geeft een TreeMap terug met twee entries. Elke entry bevat een node gelinkt aan een InetAddress. Als eerste zal er bepaald worden welke entry de next node van de gefailde node is en wie de previous node is. Dit zal gebeuren door de nodenummers te vergelijken met de gefailde node.
Zodra de eigen node weet wie de previous en next node zijn, zal hij deze contacteren met *setNextNode(int nextNode, String nextIP)* & *setPreviousNode(int previousNode, String previousIP)* via RMI om hun buur aan te passen. Natuurlijk zal er eerst gecontroleerd worden of dat de gefailde node niet toevallig een buur was een de eigen node, in dit geval moet er geen RMI naar uzelf gestuurd worden.

Replication

Voor replicatie willen we in het systeem alle files overal 1 keer gerepliceerd hebben staan, dit wilt zeggen dat als een node in het systeem komt, dat deze zijn lokale files stuurt naar de nodes waar de hash naar wijst (of naar zijn vorige als de hash naar zichzelf wijst).
Wanneer een node in het systeem komt, checkt hij eerst of hij alleen is, zo ja doet hij niets van replicatie. Als de node daarentegen niet alleen is, gaat hij aan de server vragen waar al zijn files moeten staan, dit aan de hand van de hash van de filenaam. Hij gaat dan via **tcp** al deze bestanden sturen naar die node. Voor deze te kunnen versturen zal de ene node de andere eerst laten weten (via RMI) dat hij een bestand gaat sturen, hierbij stuurt hij de poort die hij gebruikt en de naam van de file. De ontvangende node zet zich dan klaar zodat de zender kan beginnen sturen. We gebruiken hiervoor altijd een andere socketpoort, hierdoor kunnen we alles zo parallel mogelijk sturen zonder conflicten te krijgen (new ServerSocket(0) geeft een random beschikbare poort). (TCPReceiver.java en TCPSender.java)
Wat moeten de nodes doen die al in het systeem zitten? Enkel de **previousnode** van de bijkomende node moet iets doen, zo moet de previous checken of er lokale bestanden van hem op de nieuwe node zijn gemapt in plaats van op zichzelf. Als dit het geval is, zegt deze

node tegen zijn vorige previous dat hij dit moet verwijderen en stuurt hij het bestand door naar de nieuwe node. Ook moet de previous zijn replicated files overlopen, diegene dat dan naar de nieuwe node gemapt zijn, stuurt hij door en verwijdert hij bij zichzelf (dit is verschillend van de eigenlijke opgave maar leek ons beter, in de eigenlijke opgave moest dit bestand dan als downloadlocatie kunnen dienen in plaats van verwijderd te worden). Zo staan alle files die al in het systeem zaten nu op de juiste plaats.
Hierna willen we het systeem constant blijven **updaten** (CheckFileList.java), dit doen we door bij opstarten van de node een thread te starten die elke 30 seconden kijkt of er iets veranderd is bij onze lokale bestanden(ReplicateNewFiles() in Node.java). Als er een bestand bij is gekomen, sturen we dit bestand weer door naar de node waar de server deze file aan linkt. Als er een bestand is verwijderd, wordt dit ook verwijdert van de node waar deze gerepliceerd staat. Voor dit te checken hebben we in Node.java een arraylist aangemaakt met daarin de locale bestanden, om de 30 seconden vergelijken we deze dan met de nieuwe lijst.
Wat nu als er een node zichzelf wilt **uitschakelen**? (deleteNode() in Node.java) Deze node moet dan eerst al zijn gerepliceerde bestanden doorsturen naar zijn previousnode (en in sommige gevallen zelfs naar de previous van deze previousnode), hierbij sturen we het bestand maar ook alle informatie dat bij dit bestand hoort. Hierna moet de node overal in het systeem zijn lokale bestanden verwijderen als deze niet gedownload zijn.

Agents

Voor de agents geïmplementeerd worden moet er eerst geweten zijn wat deze moeten doen. In het project System Y zal de agent van node tot node rondgaan om een lijst op te maken met welke bestanden waar staan, als hij merkt dat er bestand bijgekomen is voegt hij deze toe en wanneer er een bestand weg is zal hij deze ook weg doen uit zijn lijst. Naast een lijst met alle bestanden introduceert hij ook een locked functie voor files. Wanneer een gebruiker beslist om een bepaald bestand te downloaden zal de agent een lock zetten op de plek vanwaar dit bestand gedownload wordt. Zo kan dit bestand niet gedownload worden op de moment dat een andere node dit bestand aan het downloaden is. Om de Agent van node te laten gaan zal deze geserialiseerd moeten worden om vervolgens te kunnen verzonden met RMI.

Wanneer we dit toepassen op ons project verkrijgen we volgende uitwerking. Om de agent te kunnen starten moeten er meer dan of juist 2 nodes aanwezig zijn in het systeem. Is dit het geval, dan zal er een nieuwe agent aangemaakt worden: *AgentFileList agent = new AgentFileList()* Die vervolgens uitgevoerd zal worden in thread. Om er voor te kunnen zorgen dat de agent aan de info van de node kan, roepen we ook de *agent.setNode(this)* functie aan. In de nieuwe klasse, genaamd **AgentFileList.java**, worden er enkele functies van de desbetreffende Node opgeroepen vanuit de run method. de opgeroepen functies zijn de volgende:

- *nodeagent.replicateNewFiles()* → Deze kijkt in de zijn lijst van lokale files of er bepaalde bestanden verwijderd of toegevoegd zijn. Om te zien of er bestanden verwijderd zijn wordt er een aparte lijst bijgehouden bij de node genaamd, templocalFiles. Wanneer de agent langs komt zal hij de bestanden die in die lijst staan verwijderen uit zijn algemene lijst en vervolgens de templocalFiles van de node ook ledigen.
- *update()* → Een functie die zorgt voor het bijwerken van de lijst
- checkLock() → deze method kijkt of er een bestand gelocked moet worden
- checkUnlock() → deze method kijkt of er een bestand geUnlocked moet worden
- *nodeagent.setTotalFileList(totalFileList)* → Wanneer deze lijst bijgewerkt is zal deze terug verzonden worden naar de node zodat ook zij deze lijst lokaal hebben staan.

Het grote struikblok bij dit deel is het kunnen verzenden van de agent via RMI. Naast het feit dat er bij AgentFileList implements Runnable moet staan, zal er zowel bij AgentFileList als bij Node implements Serializable moeten komen. Om het serializeren goed te laten verlopen moet

bij beide ook de serialVersionUID ingesteld worden, deze geven we de waarde 1L en moet ook exact hetzelfde zijn bij elkaar. Wanneer deze verschillend is zal deze bij het deserializeren een fout genereren wegens een verschillende UID.

Wanneer de agent klaar is bij de node zal een nieuwe thread gestarten genaamd *RMIStarter().* Deze zal als parameter de agent en het volgende IP meekrijgen. In de runfunctie van deze thread wordt via de nodeinterface *startAgentFileList()* weer aangeroepen maar dan op de volgende node.

Voor het implementeren van een failure agent was er spijtig genoeg geen tijd meer. We hebben ons meer gefocust op het correct werken van de voorgaande onderdelen. Failure op zich zal opgevangen worden door het eerder geïmplementeerde deel van in practica 4 bij Node Life Cycle.

Locken/Unlocken van bestanden

Er wordt gebruik gemaakt van twee velden. Een indexToLock en indexToUnlock. Standaard staan deze op -1. Als er een bestand moet gelocked worden gaat het syteem als volgt tewerk.

1. Zet de indexToLock op de index van totalFileList het bestand dat gelocked moet worden
2. Nu gaat de node wachten tot de agent deze index terug op -1 zet
3. De agent kijkt bij elke node waar hij langs komt of de indexToLock op een andere waarde dan één staat
4. Zo Ja locked hij dit bestand in zijn totalFileList
5. Daarna zet de agent de indexToLock terug op -1 en kan de node weer verder
6. De agent gaat nu verder door het hele systeem de totalFileList aanpassen waardoor dat bestand nu gelocked is in de totalFileList van het gehele systeem

Bestanden unlocken gebeurd op dezelfde manier. Hier maakt men gebruik van het veld indexToUnlock.

File Download

Met behulp van onze agents kunnen we nu alle bestanden bekijken die zich in het systeem bevinden. Om deze bestanden nu te gaan downloaden is er een nieuwe optie bijgekomen in het menu. Deze zal alle bestanden van het systeem afdrukken met hun index. Daarna zal er gevraagd worden om een index te geven van welk bestand je wilt downloaden. Als die index gegeven is kan de methode downloadFile aangeroepen worden in node. Deze methode doet het volgende:

1. Kijkt of dit bestand gelocked is. Als het bestand gelocked is krijg je een bericht dat dit bestand gelocked is
2. Als het bestand niet gelocked is dan gaat deze methode het bestand locken.
3. Dan wordt er gechecked of dit bestand zich al bevindt op de lokale node (bijvoorbeeld bij de lokale bestanden) is dit zo vraagt het systeem om goedkeuring of het bestand toch gedownload moet worden.
4. Daarna zal deze methode een RMI uitvoeren naar de node die het bestand moet verzenden waarna deze node het bestand verzendt over TCP
5. Hierna zal de node het bestand unlocken

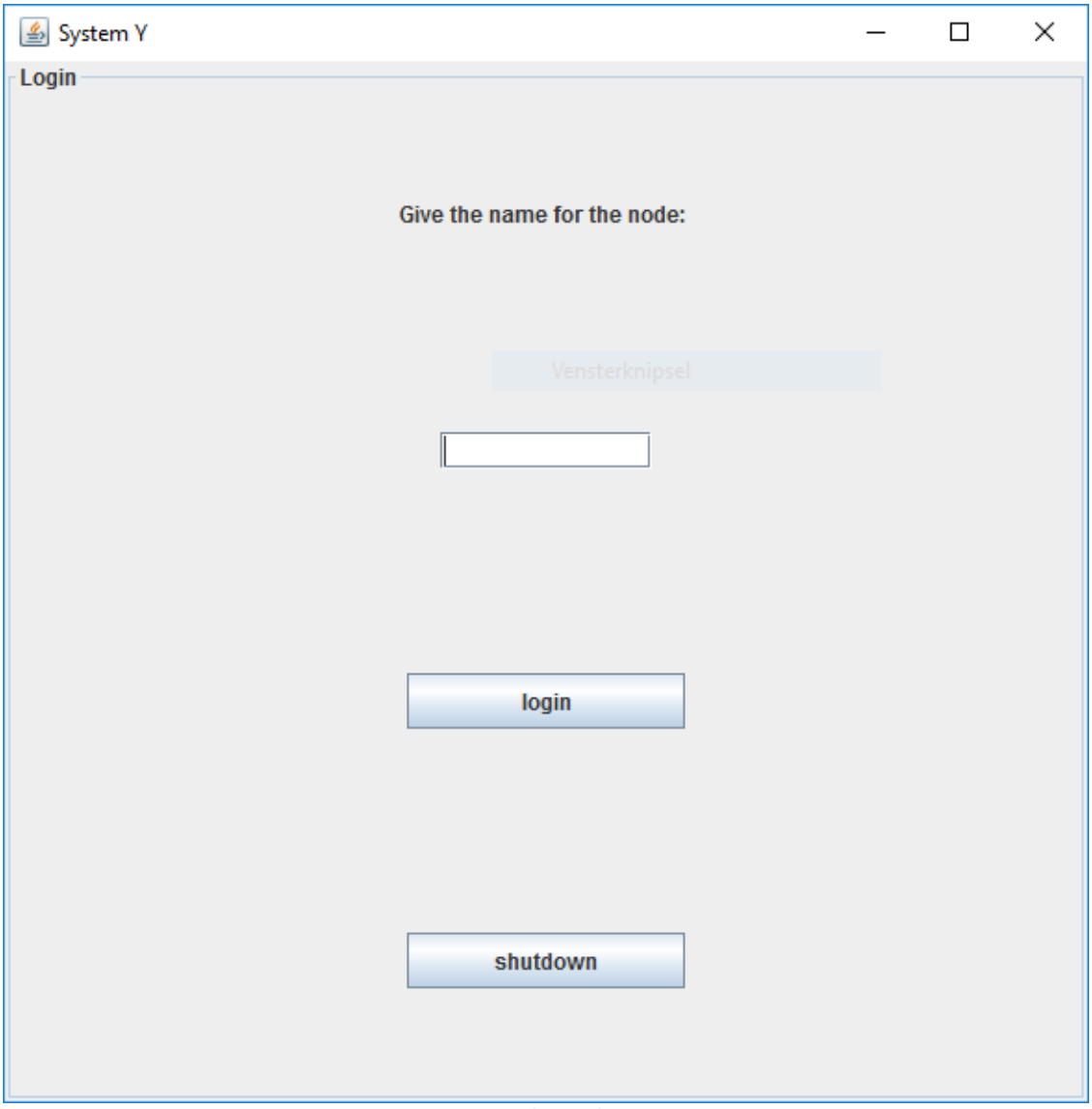
De download functionaliteit is **niet volledig**. Allereerst worden de locaties niet bijgehouden waar bestanden gedownload zijn en waar je ze van zou kunnen downloaden. Hierdoor is het

ook niet mogelijk om loadbalancing te doen om zo niet één bestand telkens van dezelfde node te downloaden. Er is dus maar één downloadlocatie.Ten tweede wordt het locken/unlocken van de bestanden in de hoofd thread uitgevoerd en omdat je het bestand pas mag unlocken op de moment dat het bestand volledig is verzonden wacht deze thread dus ook hierop waardoor je niets anders kan doen wanneer een bestand aan het downloaden is. Hier is nog ruimte voor verbetering maar spijtig genoeg geen tijd.

GUI

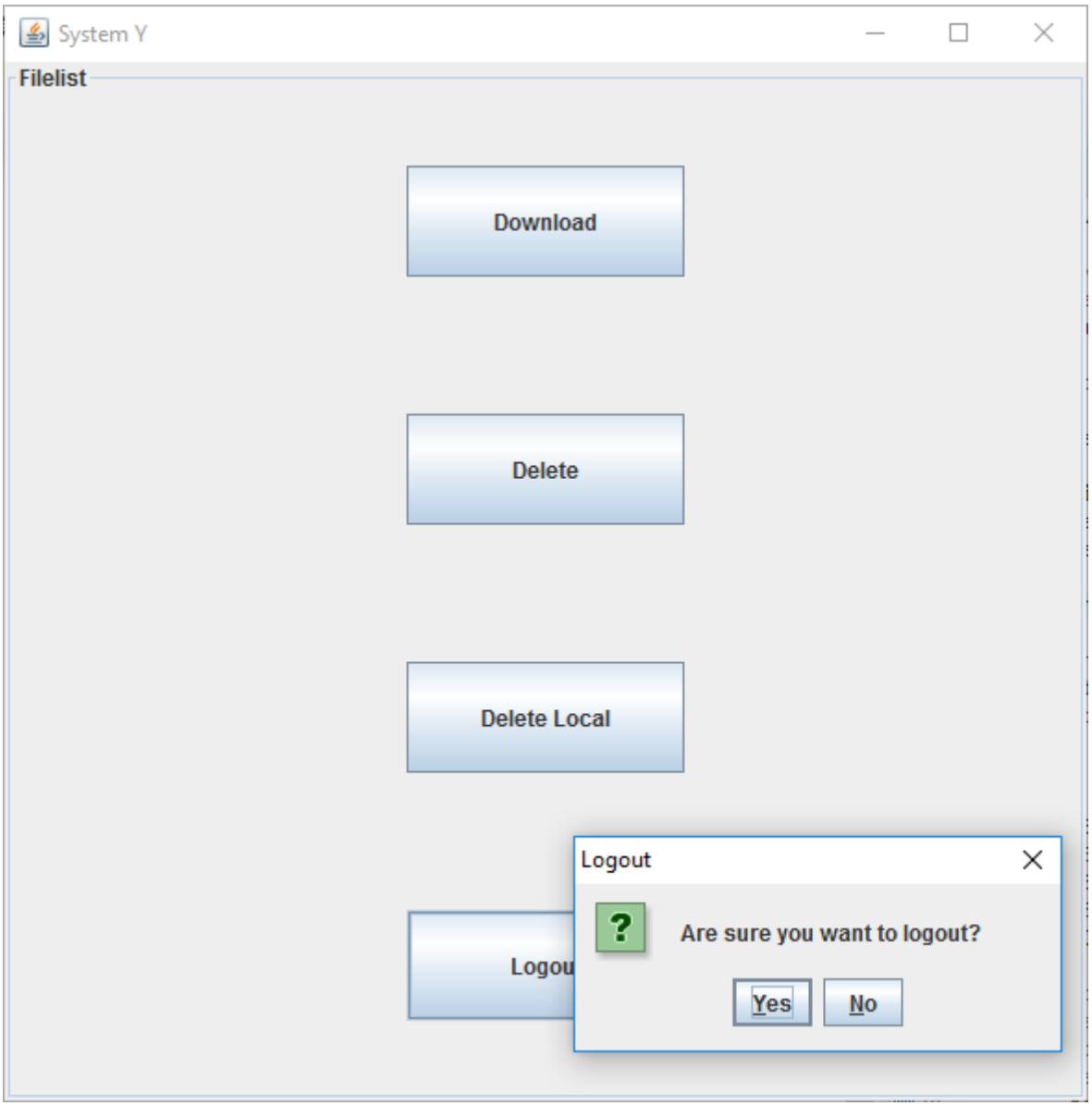
Voor het uitwerken van de GUI gingen we gebruik gemaken van het mvc-pattern. De test gui werkt (beperkt) maar door tijdgebrek hebben we dit achterwege gelaten en ervoor gezorgd dat alles wat we geïmplementeerd hebben uitvoerbaar is via de command-line. De volgende dingen werken echter ook via de GUI:

- Jezelf toevoegen in het systeem inclusief een naam voor je node geven
- Jezelf verwijderen uit het systeem



Figuur 1 loginscherm

Op bovenstaand scherm kun je de node toevoegen aan het systeem. Je geeft de naam in die je aan de node wil geven en klikt op login. Dan krijg je volgend scherm.



Figuur 2 filescherm + logout

Op dit scherm was het de bedoeling dat je via een combobox die alle bestanden bevat een bestand kan kiezen en dan één van de volgende acties uit te voeren. Je kan de node wel verwijderen uit het systeem door op de knop logout te klikken. Er zal dan om bevestiging gevraagd worden. Druk je op yes dan zal je node uit het systeem verwijderd worden.

Besluit

Uit dit project kunnen we leren dat er veel tijd moet gespendeerd worden om een goed gedistribueerd te ontwikkelen. Om ervoor te zorgen dat alles tot in de puntjes werkt kunnen we nog een tijdje bezig zijn. Toch werkt een groot deel van de basisfunctionaliteit van het systeem. Alle bestanden worden op de juiste manier gerepliceerd en elke node kan bestanden downloaden van andere nodes. Ook hebben we bij het uitwerken van dit systeem vele aspecten van gedistribueerde systemen ontdekt en toegepast. Zo hebben we gewerkt met TCP om onze bestanden te verzenden. We hebben gebruik gemaakt van multicast en RMI. Ook hebben we anders leren werken met exceptions, als een exception opkomt de juiste handeling uitvoeren, bv Failure van de Node Life Cycle. Door middel van hashing werden de bestanden quasi gelijkmatig gerepliceerd over de verschillende nodes. Tot slot hebben we agents ingevoerd in ons systeem. Deze vervullen verschillende functies zoals het maken van een lijst van bestanden en het locken of unlocken van bestanden.

Link github repository

Onderstaande link brengt u naar de repository waar ons project staat
https://github.com/lennertvanhasselt/Distributed

Bronnen

- https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html
- http://javabeat.net/introduction-to-java-agents/

Appendix: UML klasse diagram