

Akka FileSharing

Francesco Paolo Culcasi
Alessandro Martinelli
Nicola Messina

M. Sc. in Computer Engineering
Concurrent and Distributed Systems

Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 Model
 - Working principles
 - Actor scheme
- 3 Implementation
 - File Transfer
 - File Import/Creation
 - Application Logic and GUI
 - Actor system shutdown
- 4 Robustness handling
 - Node Crash

Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 Model
 - Working principles
 - Actor scheme
- 3 Implementation
 - File Transfer
 - File Import/Creation
 - Application Logic and GUI
 - Actor system shutdown
- 4 Robustness handling
 - Node Crash

System overview

- **Definition:** this software let a certain number of nodes to collaborate (forming a *cluster*) for reading and writing a group of files distributed among the nodes participating to the cluster.
- **Space:** every node dedicates a certain amount of space to the application.
- **Adding a file:** each user can add a file to the cluster either importing it from his local hard disk or by creating a new one.
- **Owner of a file:** a file added to the cluster doesn't belong indefinitely to someone; instead, at a given instant it belong to the node who own it in that moment.
- **Search of a file:** the search of a file is a tag-based search, so a certain number of tags must be associated to each file.

System overview

- **Definition:** this software let a certain number of nodes to collaborate (forming a *cluster*) for reading and writing a group of files distributed among the nodes participating to the cluster.
- **Space:** every node dedicates a certain amount of space to the application.
- **Adding a file:** each user can add a file to the cluster either importing it from his local hard disk or by creating a new one.
- **Owner of a file:** a file added to the cluster doesn't belong indefinitely to someone; instead, at a given instant it belong to the node who own it in that moment.
- **Search of a file:** the search of a file is a tag-based search, so a certain number of tags must be associated to each file.

System overview

- **Definition:** this software let a certain number of nodes to collaborate (forming a *cluster*) for reading and writing a group of files distributed among the nodes participating to the cluster.
- **Space:** every node dedicates a certain amount of space to the application.
- **Adding a file:** each user can add a file to the cluster either importing it from his local hard disk or by creating a new one.
- **Owner of a file:** a file added to the cluster doesn't belong indefinitely to someone; instead, at a given instant it belong to the node who own it in that moment.
- **Search of a file:** the search of a file is a tag-based search, so a certain number of tags must be associated to each file.

System overview

- **Definition:** this software let a certain number of nodes to collaborate (forming a *cluster*) for reading and writing a group of files distributed among the nodes participating to the cluster.
- **Space:** every node dedicates a certain amount of space to the application.
- **Adding a file:** each user can add a file to the cluster either importing it from his local hard disk or by creating a new one.
- **Owner of a file:** a file added to the cluster doesn't belong indefinitely to someone; instead, at a given instant it belong to the node who own it in that moment.
- **Search of a file:** the search of a file is a tag-based search, so a certain number of tags must be associated to each file.

System overview

- **Definition:** this software let a certain number of nodes to collaborate (forming a *cluster*) for reading and writing a group of files distributed among the nodes participating to the cluster.
- **Space:** every node dedicates a certain amount of space to the application.
- **Adding a file:** each user can add a file to the cluster either importing it from his local hard disk or by creating a new one.
- **Owner of a file:** a file added to the cluster doesn't belong indefinitely to someone; instead, at a given instant it belong to the node who own it in that moment.
- **Search of a file:** the search of a file is a tag-based search, so a certain number of tags must be associated to each file.

Index

- 1 **Requirements and assumptions**
 - What the software must do
 - **Requirements**
 - Assumptions
- 2 **Model**
 - Working principles
 - Actor scheme
- 3 **Implementation**
 - File Transfer
 - File Import/Creation
 - Application Logic and GUI
 - Actor system shutdown
- 4 **Robustness handling**
 - Node Crash

Requirements

- **Fully decentralized system:** no centralized, high-performance entity storing all the files or performing special tasks should be used.
- **Concurrent file access:** concurrent file access is handled in a **Single Writer Multiple Reader** way: in a given moment, whatever number of peer can read a certain file. Instead, if a peer is writing a file, no one else is allowed to reading or modifying the same file.
- **Fault handling:** the system should remain in a consistent state even if the following problems arise:
 - network failure
 - peer failure
- **Relaxed load balancing:** files are homogeneously distributed between the participating peers. However, it is not required that in every instant the load is distributed in the most fair way possible, instead this may be considered a general guideline.

Requirements

- **Fully decentralized system:** no centralized, high-performance entity storing all the files or performing special tasks should be used.
- **Concurrent file access:** concurrent file access is handled in a **Single Writer Multiple Reader** way: in a given moment, whatever number of peer can read a certain file. Instead, if a peer is writing a file, no one else is allowed to reading or modifying the same file.
- **Fault handling:** the system should remain in a consistent state even if the following problems arise:
 - network failure
 - peer failure
- **Relaxed load balancing:** files are homogeneously distributed between the participating peers. However, it is not required that in every instant the load is distributed in the most fair way possible, instead this may be considered a general guideline.

Requirements

- **Fully decentralized system:** no centralized, high-performance entity storing all the files or performing special tasks should be used.
- **Concurrent file access:** concurrent file access is handled in a **Single Writer Multiple Reader** way: in a given moment, whatever number of peer can read a certain file. Instead, if a peer is writing a file, no one else is allowed to reading or modifying the same file.
- **Fault handling:** the system should remain in a consistent state even if the following problems arise:
 - network failure
 - peer failure
- **Relaxed load balancing:** files are homogeneously distributed between the participating peers. However, it is not required that in every instant the load is distributed in the most fair way possible, instead this may be considered a general guideline.

Requirements

- **Fully decentralized system:** no centralized, high-performance entity storing all the files or performing special tasks should be used.
- **Concurrent file access:** concurrent file access is handled in a **Single Writer Multiple Reader** way: in a given moment, whatever number of peer can read a certain file. Instead, if a peer is writing a file, no one else is allowed to reading or modifying the same file.
- **Fault handling:** the system should remain in a consistent state even if the following problems arise:
 - network failure
 - peer failure
- **Relaxed load balancing:** files are homogeneously distributed between the participating peers. However, it is not required that in every instant the load is distributed in the most fair way possible, instead this may be considered a general guideline.

Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 Model
 - Working principles
 - Actor scheme
- 3 Implementation
 - File Transfer
 - File Import/Creation
 - Application Logic and GUI
 - Actor system shutdown
- 4 Robustness handling
 - Node Crash

Assumptions

- **File editing:** in order for a peer to modifying a file, the file must be located in the peer's local space.
An user may modify up to one file at a time.
At the end of the modification, a load balancing algorithm is executed for deciding where to put the file (in which node).
- **File reading:** files to read are stored in the *temp* folder, so they don't fill the application's available space.
- **Leave of a node:** when a node leaves voluntarily, both files and informations located in his local space must remain reachable: a load balancing algorithm and an information redistribution algorithm will be executed.
- **File name:** the name of a file must be unique within the entire cluster. A file name is considered the same way as a tag.

Assumptions

- **File editing:** in order for a peer to modifying a file, the file must be located in the peer's local space.
An user may modify up to one file at a time.
At the end of the modification, a load balancing algorithm is executed for deciding where to put the file (in which node).
- **File reading:** files to read are stored in the *temp* folder, so they don't fill the application's available space.
- **Leave of a node:** when a node leaves voluntarily, both files and informations located in his local space must remain reachable: a load balancing algorithm and an information redistribution algorithm will be executed.
- **File name:** the name of a file must be unique within the entire cluster. A file name is considered the same way as a tag.

Assumptions

- **File editing:** in order for a peer to modifying a file, the file must be located in the peer's local space.
An user may modify up to one file at a time.
At the end of the modification, a load balancing algorithm is executed for deciding where to put the file (in which node).
- **File reading:** files to read are stored in the *temp* folder, so they don't fill the application's available space.
- **Leave of a node:** when a node leaves voluntarily, both files and informations located in his local space must remain reachable: a load balancing algorithm and an information redistribution algorithm will be executed.
- **File name:** the name of a file must be unique within the entire cluster. A file name is considered the same way as a tag.

Assumptions

- **File editing:** in order for a peer to modifying a file, the file must be located in the peer's local space.
An user may modify up to one file at a time.
At the end of the modification, a load balancing algorithm is executed for deciding where to put the file (in which node).
- **File reading:** files to read are stored in the *temp* folder, so they don't fill the application's available space.
- **Leave of a node:** when a node leaves voluntarily, both files and informations located in his local space must remain reachable: a load balancing algorithm and an information redistribution algorithm will be executed.
- **File name:** the name of a file must be unique within the entire cluster. A file name is considered the same way as a tag.

Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 Model
 - **Working principles**
 - Actor scheme
- 3 Implementation
 - File Transfer
 - File Import/Creation
 - Application Logic and GUI
 - Actor system shutdown
- 4 Robustness handling
 - Node Crash

Locating files

- Files are distributed among cluster's nodes, with the policy *"a file who has to be added to the system goes to the node with higher free space"*.
- Since files have not a fixed residing node we need informations to locate them.
- Using **Hash tables** let us obtain a quicker lookup and insertion of the information about the owner of a file (constant-time). We need a key-value association through hash function (or something similar, i.e. id key="file name/tag" value="node responsible for the information").
- We don't want the whole hash table to reside in a single node.
⇒ **Distributed Hash Table**
Files are spread across multiple nodes, with each node taking *responsibility* for a portion of the key-space.

Locating files

- Files are distributed among cluster's nodes, with the policy *"a file who has to be added to the system goes to the node with higher free space"*.
- Since files have not a fixed residing node we need informations to locate them.
- Using **Hash tables** let us obtain a quicker lookup and insertion of the information about the owner of a file (constant-time). We need a key-value association through hash function (or something similar, i.e. id key="file name/tag" value="node responsible for the information").
- We don't want the whole hash table to reside in a single node.
⇒ **Distributed Hash Table**
Files are spread across multiple nodes, with each node taking *responsibility* for a portion of the key-space.

Locating files

- Files are distributed among cluster's nodes, with the policy *"a file who has to be added to the system goes to the node with higher free space"*.
- Since files have not a fixed residing node we need informations to locate them.
- Using **Hash tables** let us obtain a quicker lookup and insertion of the information about the owner of a file (constant-time). We need a key-value association through hash function (or something similar, i.e. id key="file name/tag" value="node responsible for the information").
- We don't want the whole hash table to reside in a single node.
⇒ **Distributed Hash Table**
Files are spread across multiple nodes, with each node taking *responsibility* for a portion of the key-space.

Locating files

- Files are distributed among cluster's nodes, with the policy *"a file who has to be added to the system goes to the node with higher free space"*.
- Since files have not a fixed residing node we need informations to locate them.
- Using **Hash tables** let us obtain a quicker lookup and insertion of the information about the owner of a file (constant-time). We need a key-value association through hash function (or something similar, i.e. id key="file name/tag" value="node responsible for the information").
- We don't want the whole hash table to reside in a single node.
⇒ **Distributed Hash Table**
Files are spread across multiple nodes, with each node taking *responsibility* for a portion of the key-space.

The Chord Protocol

Problems

- How to figure out which node is responsible for a file?
- How to handle changes to the network topology? (Nodes can join or leave the cluster)

Solution

The Chord Protocol

- Chord¹ is a protocol and a set of algorithms for implementing a distributed hash table.
- Does not prescribe replication techniques.
- Redistribution of data associated with key when nodes leave or join the network it's up to the application.

¹Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

The Chord Protocol

Problems

- How to figure out which node is responsible for a file?
- How to handle changes to the network topology? (Nodes can join or leave the cluster)

Solution

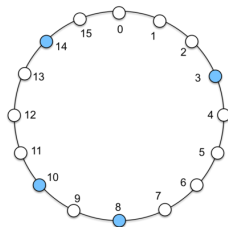
The Chord Protocol

- Chord¹ is a protocol and a set of algorithms for implementing a distributed hash table.
- Does not prescribe replication techniques.
- Redistribution of data associated with key when nodes leave or join the network it's up to the application.

¹Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications

Chord in action 1/3

- The key-space can be visualized as a ring.
- A function $f()$ (we'll see later) is used to map keys onto location on the ring.
- Nodes are also mapped to locations on the ring (determined by applying $f()$ that maps their IP address and port number to unique IDs).



Chord in action 2/3

- **Consistent Hashing scheme:**
Chord assigns responsibility for segments of the ring to individual nodes.
It allows them to be added or removed from the cluster while minimizing the number of key that will need to be reassigned.
- Given a file and the corresponding key (obtained applying $f()$ to one of its tags) the node responsible for the information "*which node owns the file*" is the one whose location on the ring is equal or greater than the key.

Chord in action 2/3

- **Consistent Hashing scheme:**
Chord assigns responsibility for segments of the ring to individual nodes.
It allows them to be added or removed from the cluster while minimizing the number of key that will need to be reassigned.
- Given a file and the corresponding key (obtained applying $f()$ to one of its tags) the node responsible for the information "*which node owns the file*" is the one whose location on the ring is equal or greater than the key.

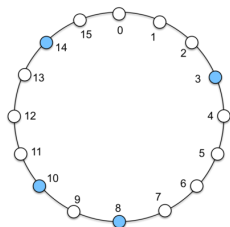
Chord in action 2/3

- **Consistent Hashing scheme:**

Chord assigns responsibility for segments of the ring to individual nodes.

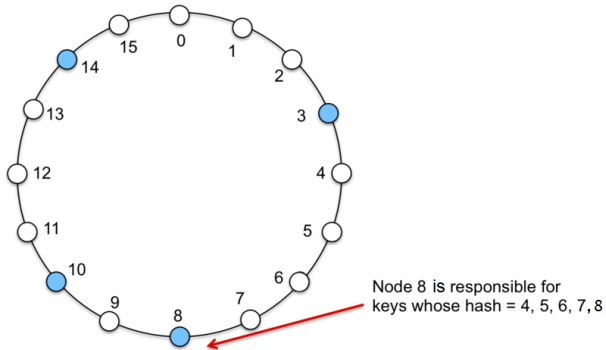
It allows them to be added or removed from the cluster while minimizing the number of key that will need to be reassigned.

- Given a file and the corresponding key (obtained applying $f()$ to one of its tags) the node responsible for the information *"which node owns the file"* is the one whose location on the ring is equal or greater than the key.



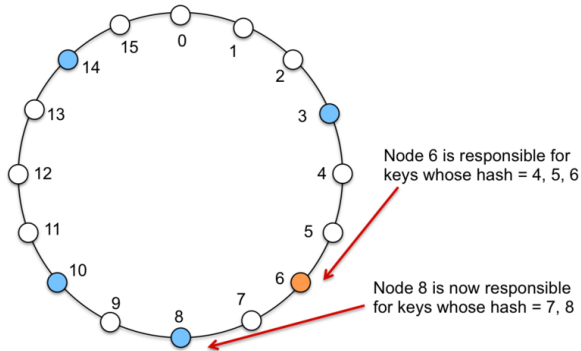
```
file: "foo.txt"
owner: node14
tags: "foo.txt", "foo", "bar", ...
f("foo") returns 5
f(131.112.193.73:2556) returns 8
↓
node8 is responsible for tag "foo"
node8 says «"foo.txt" is in node14»
```

Chord in action 3/3



Present nodes are the blue circles

Chord in action 3/3



Node 6 has been added to the network

Hash collisions problem

- Could be Hash function a good $f()$ function?
- **Problem:** hash functions generate collisions
- How to identify uniquely a node? Using a block cipher (e.g. AES) with fixed key and IV.
 - Uniform distribution of IDs.
 - **Bijective.**
- AES translates IP + port (6 bytes) in fixed length 16 bytes block \implies IDs are 16 bytes wide.
- We use AES also for the tags even if it's not strictly necessary since tag collisions are handled properly by means of the FileInfoTable.

Hash collisions problem

- Could be Hash function a good $f()$ function?
- **Problem:** hash functions generate collisions
- How to identify uniquely a node? Using a block cipher (e.g. AES) with fixed key and IV.
 - Uniform distribution of IDs.
 - **Bijective.**
- AES translates IP + port (6 bytes) in fixed length 16 bytes block \implies IDs are 16 bytes wide.
- We use AES also for the tags even if it's not strictly necessary since tag collisions are handled properly by means of the FileInfoTable.

Hash collisions problem

- Could be Hash function a good $f()$ function?
- **Problem:** hash functions generate collisions
- How to identify uniquely a node? Using a block cipher (e.g. AES) with fixed key and IV.
 - Uniform distribution of IDs.
 - **Bijective.**
- AES translates IP + port (6 bytes) in fixed length 16 bytes block \implies IDs are 16 bytes wide.
- We use AES also for the tags even if it's not strictly necessary since tag collisions are handled properly by means of the FileInfoTable.

Hash collisions problem

- Could be Hash function a good $f()$ function?
- **Problem:** hash functions generate collisions
- How to identify uniquely a node? Using a block cipher (e.g. AES) with fixed key and IV.
 - Uniform distribution of IDs.
 - **Bijective.**
- AES translates IP + port (6 bytes) in fixed length 16 bytes block \implies IDs are 16 bytes wide.
- We use AES also for the tags even if it's not strictly necessary since tag collisions are handled properly by means of the FileInfoTable.

Node8's FileInfoTable	
jones	LittleJones.pdf, owner=18 IndianaJones.avi, owner=7
foo	Foo.txt, owner=14 FooFighter-Walk.mp3, owner=20
Indiana	IndianaJones.avi, owner=7 IndianaEvans.jpg, owner=2
....	
Foo.txt	Foo.txt, owner=14

Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 **Model**
 - Working principles
 - **Actor scheme**
- 3 Implementation
 - File Transfer
 - File Import/Creation
 - Application Logic and GUI
 - Actor system shutdown
- 4 Robustness handling
 - Node Crash

Akka Actors

Each node is composed by the following actors:

- | | |
|--------------------------|---|
| ClusterListener | To interface the node with the cluster. It stores DHT infos which are under responsibility of the corresponding node, according to the chord protocol. |
| ServerActor | To manage the list of the files residing on the node and state informations guaranteeing data consistency. It also manages file requests through TCP connections. |
| FileTransferActor | To issue file requests and to answer requests. |
| SoulReaper | To monitor the other actors and to supervise the correct closing procedure of the system. |
| GUIActor | To handle GUI events and updates in an asynchronous way. GUIActor runs on JavaFX thread. |

Akka Actors

Each node is composed by the following actors:

- | | |
|--------------------------|---|
| ClusterListener | To interface the node with the cluster. It stores DHT infos which are under responsibility of the corresponding node, according to the chord protocol. |
| ServerActor | To manage the list of the files residing on the node and state informations guaranteeing data consistency. It also manages file requests through TCP connections. |
| FileTransferActor | To issue file requests and to answer requests. |
| SoulReaper | To monitor the other actors and to supervise the correct closing procedure of the system. |
| GUIActor | To handle GUI events and updates in an asynchronous way. GUIActor runs on JavaFX thread. |

Akka Actors

Each node is composed by the following actors:

- | | |
|--------------------------|---|
| ClusterListener | To interface the node with the cluster. It stores DHT infos which are under responsibility of the corresponding node, according to the chord protocol. |
| ServerActor | To manage the list of the files residing on the node and state informations guaranteeing data consistency. It also manages file requests through TCP connections. |
| FileTransferActor | To issue file requests and to answer requests. |
| SoulReaper | To monitor the other actors and to supervise the correct closing procedure of the system. |
| GUIActor | To handle GUI events and updates in an asynchronous way. GUIActor runs on JavaFX thread. |

Akka Actors

Each node is composed by the following actors:

- | | |
|--------------------------|---|
| ClusterListener | To interface the node with the cluster. It stores DHT infos which are under responsibility of the corresponding node, according to the chord protocol. |
| ServerActor | To manage the list of the files residing on the node and state informations guaranteeing data consistency. It also manages file requests through TCP connections. |
| FileTransferActor | To issue file requests and to answer requests. |
| SoulReaper | To monitor the other actors and to supervise the correct closing procedure of the system. |
| GUIActor | To handle GUI events and updates in an asynchronous way. GUIActor runs on JavaFX thread. |

Akka Actors


Each node is composed by the following actors:

- | | |
|--------------------------|---|
| ClusterListener | To interface the node with the cluster. It stores DHT infos which are under responsibility of the corresponding node, according to the chord protocol. |
| ServerActor | To manage the list of the files residing on the node and state informations guaranteeing data consistency. It also manages file requests through TCP connections. |
| FileTransferActor | To issue file requests and to answer requests. |
| SoulReaper | To monitor the other actors and to supervise the correct closing procedure of the system. |
| GUIActor | To handle GUI events and updates in an asynchronous way. GUIActor runs on JavaFX thread. |


Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 Model
 - Working principles
 - Actor scheme
- 3 **Implementation**
 - File Transfer
 - File Import/Creation
 - Application Logic and GUI
 - Actor system shutdown
- 4 Robustness handling
 - Node Crash


Cluster System

- Use of  akka, an open-source actor-based toolkit.
- Use of Akka Cluster, that provides (as far as we are concerned):
 - Decentralized P2P gossip-based cluster membership.
 - Automatic fail-over upon node crash: the cluster sees that node as
 - *Member unreachable* the first time it is impossible to contact him and
 - *Member removed* after 10 seconds since it was marked as unreachable.
 - A leaving member is seen by the cluster as:
 - *Member leaving* as soon as the `cluster.leave()` method has been invoked.
 - *Member removed* immediately after it was marked as member leaving.


Cluster System

- Use of  akka, an open-source actor-based toolkit.
- Use of Akka Cluster, that provides (as far as we are concerned):
 - Decentralized P2P gossip-based cluster membership.
 - Automatic fail-over upon node crash: the cluster sees that node as
 - *Member unreachable* the first time it is impossible to contact him and
 - *Member removed* after 10 seconds since it was marked as unreachable.
 - A leaving member is seen by the cluster as:
 - *Member leaving* as soon as the `cluster.leave()` method has been invoked.
 - *Member removed* immediately after it was marked as member leaving.


Cluster System

- Use of  akka, an open-source actor-based toolkit.
- Use of Akka Cluster, that provides (as far as we are concerned):
 - Decentralized P2P gossip-based cluster membership.
 - Automatic fail-over upon node crash: the cluster sees that node as
 - *Member unreachable* the first time it is impossible to contact him and
 - *Member removed* after 10 seconds since it was marked as unreachable.
 - A leaving member is seen by the cluster as:
 - *Member leaving* as soon as the `cluster.leave()` method has been invoked.
 - *Member removed* immediately after it was marked as member leaving.

Cluster System

- Use of  akka, an open-source actor-based toolkit.
- Use of Akka Cluster, that provides (as far as we are concerned):
 - Decentralized P2P gossip-based cluster membership.
 - Automatic fail-over upon node crash: the cluster sees that node as
 - *Member unreachable* the first time it is impossible to contact him and
 - *Member removed* after 10 seconds since it was marked as unreachable.
 - A leaving member is seen by the cluster as:
 - *Member leaving* as soon as the `cluster.leave()` method has been invoked.
 - *Member removed* immediately after it was marked as member leaving.

Cluster System

- Use of  akka, an open-source actor-based toolkit.
- Use of Akka Cluster, that provides (as far as we are concerned):
 - Decentralized P2P gossip-based cluster membership.
 - Automatic fail-over upon node crash: the cluster sees that node as
 - *Member unreachable* the first time it is impossible to contact him and
 - *Member removed* after 10 seconds since it was marked as unreachable.
 - A leaving member is seen by the cluster as:
 - *Member leaving* as soon as the `cluster.leave()` method has been invoked.
 - *Member removed* immediately after it was marked as member leaving.

Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 Model
 - Working principles
 - Actor scheme
- 3 **Implementation**
 - **File Transfer**
 - File Import/Creation
 - Application Logic and GUI
 - Actor system shutdown
- 4 Robustness handling
 - Node Crash

File Transfer overview

- A FileTransferActor (FTA) is instantiated to handle a transfer instance.
- The FTA activity is transparent to the user (during normal load balancing operation), unless the user itself asked explicitly for a transfer (read/modify event).
- FTA orthogonal behaviors:
 - Responder: works as server that listen for incoming transfer requests.
 - Asker: works as client that initiate the transfer protocol.

	Asker	Responder
Sender	"I want send you a file"	"Sure! I'll send you a file"
Receiver	"Can you send me that file?"	"I'm ready to receive the file"

File Transfer overview

- A FileTransferActor (FTA) is instantiated to handle a transfer instance.
- The FTA activity is transparent to the user (during normal load balancing operation), unless the user itself asked explicitly for a transfer (read/modify event).
- FTA orthogonal behaviors:
 - Responder: works as server that listen for incoming transfer requests.
 - Asker: works as client that initiate the transfer protocol.

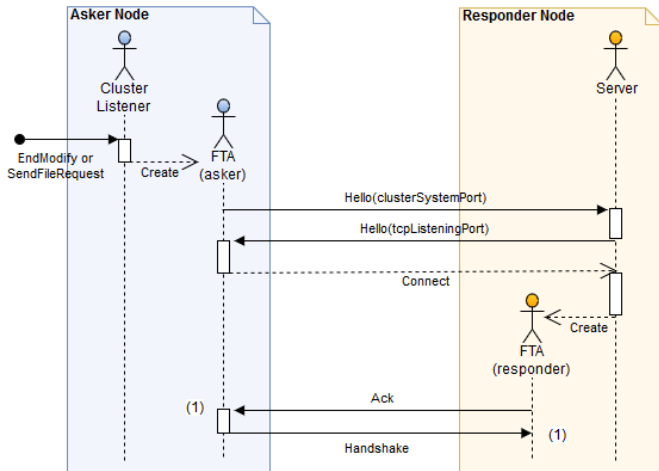
	Asker	Responder
Sender	"I want send you a file"	"Sure! I'll send you a file"
Receiver	"Can you send me that file?"	"I'm ready to receive the file"

File Transfer overview

- A FileTransferActor (FTA) is instantiated to handle a transfer instance.
- The FTA activity is transparent to the user (during normal load balancing operation), unless the user itself asked explicitly for a transfer (read/modify event).
- FTA orthogonal behaviors:
 - Responder: works as server that listen for incoming transfer requests.
 - Asker: works as client that initiate the transfer protocol.

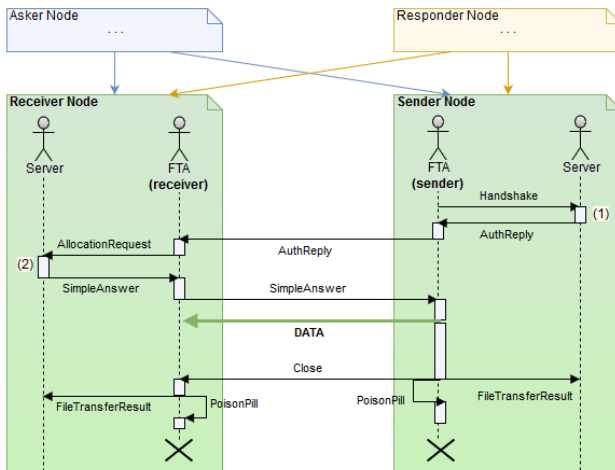
	Asker	Responder
Sender	"I want send you a file"	"Sure! I'll send you a file"
Receiver	"Can you send me that file?"	"I'm ready to receive the file"

File Transfer - FTA Initialization



(1) The FTAs behaviors are specialized: one will be **sender** and the other **receiver**, depending on the Asker request

File Transfer - Trasfer protocol



- (1) The Server checks if the file is unlocked. If so, it is locked to be transferred
(2) The Server checks if there is enough space in order to contain the new file. If so, the Server allocates the entry in the FileTable in locked mode.

FTAs behavior

- Load Balancing request: FTA in asker sender mode
- Read/Modify request: FTA in asker receiver mode
- Potential issues
 - Message stash and change behavior techniques to handle multiple connections at the same time.
 - Potential bottleneck problem on the server when it is asked for multiple transfers.
- Modify protocol:
 - So far we've seen the file transfer only; actually, when an user click *modify*, other things happen, as we will see in the next slide.

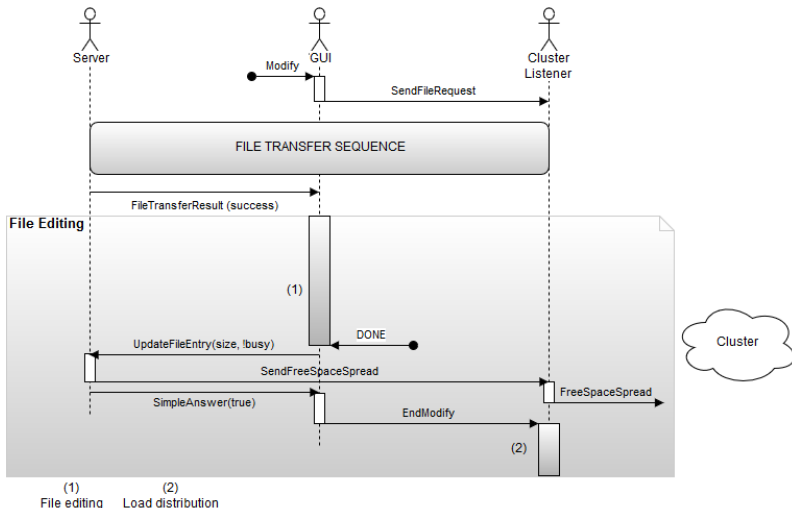
FTAs behavior

- Load Balancing request: FTA in asker sender mode
- Read/Modify request: FTA in asker receiver mode
- Potential issues
 - Message stash and change behavior techniques to handle multiple connections at the same time.
 - Potential bottleneck problem on the server when it is asked for multiple transfers.
- Modify protocol:
 - So far we've seen the file transfer only; actually, when an user click *modify*, other things happen, as we will see in the next slide.

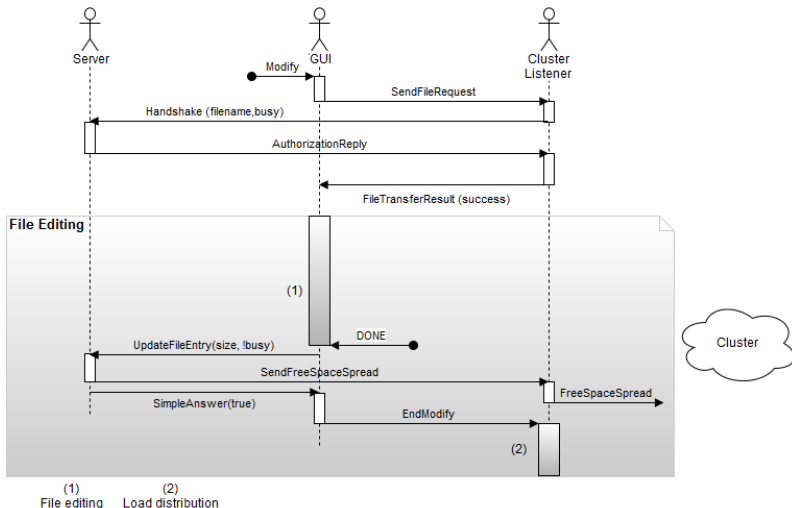
FTAs behavior

- Load Balancing request: FTA in asker sender mode
- Read/Modify request: FTA in asker receiver mode
- Potential issues
 - Message stash and change behavior techniques to handle multiple connections at the same time.
 - Potential bottleneck problem on the server when it is asked for multiple transfers.
- Modify protocol:
 - So far we've seen the file transfer only; actually, when an user click *modify*, other things happen, as we will see in the next slide.

The Modify case



The modify case (the sender is myself)



Rollback

- During the transfer something bad (node crash, link failure) may happen.
- In that case rollback must be performed in order to guarantee a consistent view of the system.
- Receiver rollback obeys:
 - Deleting the file entry from the file table in the Server.
 - Deleting the file.
 - Restoring the free space.
- Sender rollback obeys:
 - Releasing the lock on the file.

Rollback

- During the transfer something bad (node crash, link failure) may happen.
- In that case rollback must be performed in order to guarantee a consistent view of the system.
- Receiver rollback obeys:
 - Deleting the file entry from the file table in the Server.
 - Deleting the file.
 - Restoring the free space.
- Sender rollback obeys:
 - Releasing the lock on the file.

Rollback

- During the transfer something bad (node crash, link failure) may happen.
- In that case rollback must be performed in order to guarantee a consistent view of the system.
- Receiver rollback obeys:
 - Deleting the file entry from the file table in the Server.
 - Deleting the file.
 - Restoring the free space.
- Sender rollback obeys:
 - Releasing the lock on the file.

Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 Model
 - Working principles
 - Actor scheme
- 3 **Implementation**
 - File Transfer
 - **File Import/Creation**
 - Application Logic and GUI
 - Actor system shutdown
- 4 Robustness handling
 - Node Crash

File Import/Creation overview

- File name uniqueness must be guaranteed at every time
 - The Cluster Listener Actor handles file creation in a safe way.
 - On reception of a creation request, the file name tag is potentially reserved through a testAndSet mechanism
- Creation and Import are handled mostly the same way
 - The difference is how the file is created.
- Let's see how the Creation works

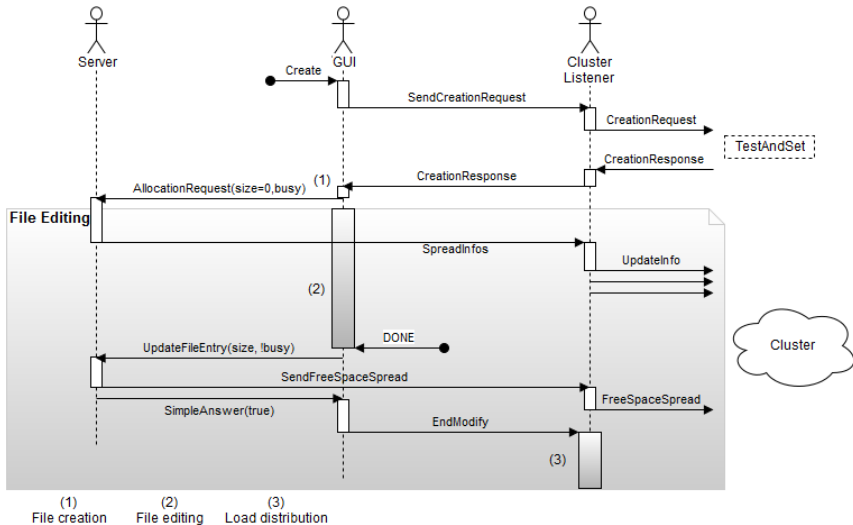
File Import/Creation overview

- File name uniqueness must be guaranteed at every time
 - The Cluster Listener Actor handles file creation in a safe way.
 - On reception of a creation request, the file name tag is potentially reserved through a testAndSet mechanism
- Creation and Import are handled mostly the same way
 - The difference is how the file is created.
- Let's see how the Creation works

File Import/Creation overview

- File name uniqueness must be guaranteed at every time
 - The Cluster Listener Actor handles file creation in a safe way.
 - On reception of a creation request, the file name tag is potentially reserved through a testAndSet mechanism
- Creation and Import are handled mostly the same way
 - The difference is how the file is created.
- Let's see how the Creation works

The Creation case



Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 Model
 - Working principles
 - Actor scheme
- 3 **Implementation**
 - File Transfer
 - File Import/Creation
 - **Application Logic and GUI**
 - Actor system shutdown
- 4 Robustness handling
 - Node Crash

Application Logic and GUI overview

- **Problem:** The GUI receives asynchronous updates from multiple system actors. They must be handled concurrently.
- Possible solutions:
 - Pass GUI object references to the actors in a thread safe way.
 - Follow Akka design methods and provide an Akka Actor also to handle GUI.
- With the latest solution some problem could anyway arise if the GUI Actor is not in the same thread context used by the GUI (explicit synchronization should be used).
- **Solution:** Put the GUI Actor on the single thread dispatcher of JavaFX, overriding the default Akka dispatcher for the GUI Actor.

Application Logic and GUI overview

- **Problem:** The GUI receives asynchronous updates from multiple system actors. They must be handled concurrently.
- Possible solutions:
 - Pass GUI object references to the actors in a thread safe way.
 - Follow Akka design methods and provide an Akka Actor also to handle GUI.
- With the latest solution some problem could anyway arise if the GUI Actor is not in the same thread context used by the GUI (explicit synchronization should be used).
- **Solution:** Put the GUI Actor on the single thread dispatcher of JavaFX, overriding the default Akka dispatcher for the GUI Actor.

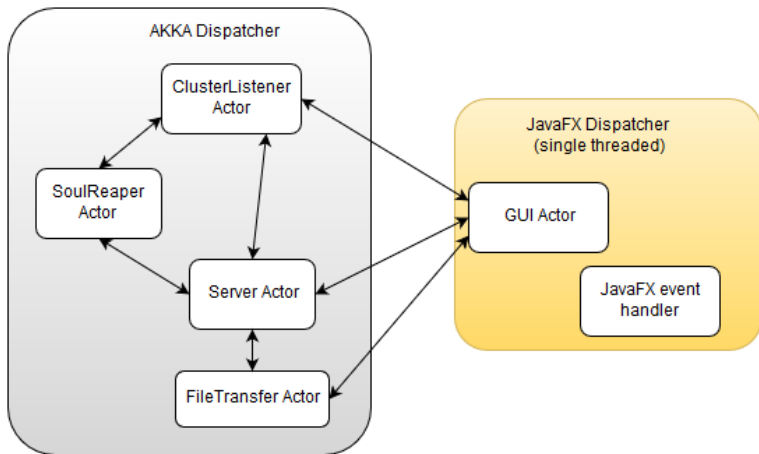
Application Logic and GUI overview

- **Problem:** The GUI receives asynchronous updates from multiple system actors. They must be handled concurrently.
- Possible solutions:
 - Pass GUI object references to the actors in a thread safe way.
 - Follow Akka design methods and provide an Akka Actor also to handle GUI.
- With the latest solution some problem could anyway arise if the GUI Actor is not in the same thread context used by the GUI (explicit synchronization should be used).
- **Solution:** Put the GUI Actor on the single thread dispatcher of JavaFX, overriding the default Akka dispatcher for the GUI Actor.

Application Logic and GUI overview

- **Problem:** The GUI receives asynchronous updates from multiple system actors. They must be handled concurrently.
- Possible solutions:
 - Pass GUI object references to the actors in a thread safe way.
 - Follow Akka design methods and provide an Akka Actor also to handle GUI.
- With the latest solution some problem could anyway arise if the GUI Actor is not in the same thread context used by the GUI (explicit synchronization should be used).
- **Solution:** Put the GUI Actor on the single thread dispatcher of JavaFX, overriding the default Akka dispatcher for the GUI Actor.

JavaFX and AKKA dispatchers



Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 Model
 - Working principles
 - Actor scheme
- 3 **Implementation**
 - File Transfer
 - File Import/Creation
 - Application Logic and GUI
 - **Actor system shutdown**
- 4 Robustness handling
 - Node Crash

SoulReaper Actor

- Shutting down an actor system must be done in a precise way, in order to let all the task to terminate properly.
- The SoulReaper Actor implements a shutdown pattern
 - Every actor in the system is watched by the SoulReaper, that keeps track of all the live actors.
 - If an actor dies (even commit suicide or it is killed by others) the SoulReaper Actor "collects his soul".
 - If all the souls are reaped, the SoulReaper terminates the entire actor system closing the application.

SoulReaper Actor

- Shutting down an actor system must be done in a precise way, in order to let all the task to terminate properly.
- The SoulReaper Actor implements a shutdown pattern
 - Every actor in the system is watched by the SoulReaper, that keeps track of all the live actors.
 - If an actor dies (even commit suicide or it is killed by others) the SoulReaper Actor "collects his soul".
 - If all the souls are reaped, the SoulReaper terminates the entire actor system closing the application.

SoulReaper Actor

- Shutting down an actor system must be done in a precise way, in order to let all the task to terminate properly.
- The SoulReaper Actor implements a shutdown pattern
 - Every actor in the system is watched by the SoulReaper, that keeps track of all the live actors.
 - If an actor dies (even commit suicide or it is killed by others) the SoulReaper Actor "collects his soul".
 - If all the souls are reaped, the SoulReaper terminates the entire actor system closing the application.

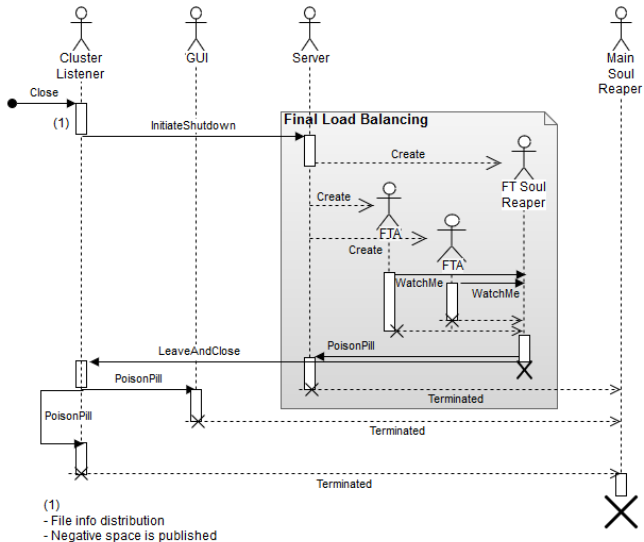
SoulReaper Actor

- Shutting down an actor system must be done in a precise way, in order to let all the task to terminate properly.
- The SoulReaper Actor implements a shutdown pattern
 - Every actor in the system is watched by the SoulReaper, that keeps track of all the live actors.
 - If an actor dies (even commit suicide or it is killed by others) the SoulReaper Actor "collects his soul".
 - If all the souls are reaped, the SoulReaper terminates the entire actor system closing the application.

SoulReaper Actor

- Shutting down an actor system must be done in a precise way, in order to let all the task to terminate properly.
- The SoulReaper Actor implements a shutdown pattern
 - Every actor in the system is watched by the SoulReaper, that keeps track of all the live actors.
 - If an actor dies (even commit suicide or it is killed by others) the SoulReaper Actor "collects his soul".
 - If all the souls are reaped, the SoulReaper terminates the entire actor system closing the application.

Shutdown procedure



Index

- 1 Requirements and assumptions
 - What the software must do
 - Requirements
 - Assumptions
- 2 Model
 - Working principles
 - Actor scheme
- 3 Implementation
 - File Transfer
 - File Import/Creation
 - Application Logic and GUI
 - Actor system shutdown
- 4 Robustness handling
 - Node Crash

- Filename tag renewal in case of responsible node crash
 - This case the node that owns the file is responsible for the filename tag renewal.
 - The other tags are supposed to be lost in case of a large crash.
 - This is an acceptable behavior, since the tags regarding a file are spread all over the cluster, not on a single node.
- FileTable image is saved on disk
 - If a node crashes, the files owned are no more available.
 - However, serializing the FileTable on disk, their state can be reloaded into the application once the application is restarted.
 - This solution doesn't scale to disk failures.

- Filename tag renewal in case of responsible node crash
 - This case the node that owns the file is responsible for the filename tag renewal.
 - The other tags are supposed to be lost in case of a large crash.
 - This is an acceptable behavior, since the tags regarding a file are spread all over the cluster, not on a single node.
- FileTable image is saved on disk
 - If a node crashes, the files owned are no more available.
 - However, serializing the FileTable on disk, their state can be reloaded into the application once the application is restarted.
 - This solution doesn't scale to disk failures.