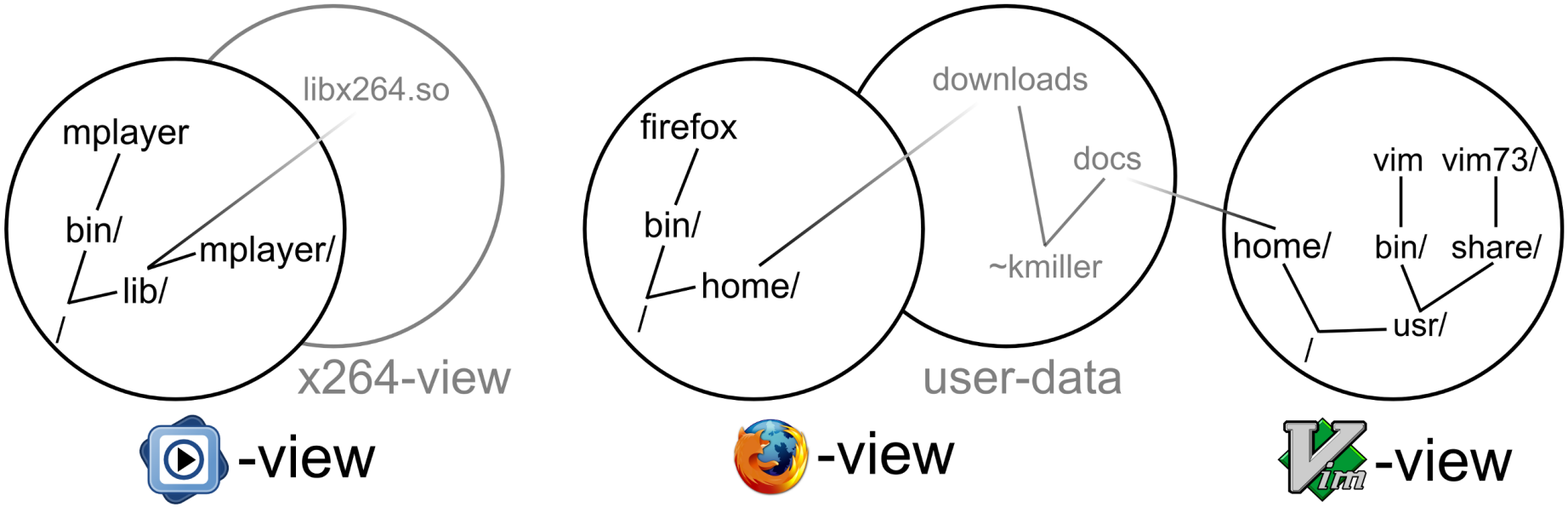
Karlsruhe Institute of Technology

A Case for Dynamic File System Views

Konrad Miller <miller@kit.edu>

KIT - System Architecture Group | EuroSys DW, April 2011



Motivation 1/2

With classical file systems and package managers it is hard or impossible to . . .

- . . . install software from different distributions side-by-side

- . . . use multiple versions of the same software at the same time

- . . . automatically fetch and replace packages on demand

Motivation 1/2

With classical file systems and package managers it is hard or impossible to . . .

- . . . install software from different distributions side-by-side

- . . . use multiple versions of the same software at the same time

- . . . automatically fetch and replace packages on demand

Motivation 1/2

With classical file systems and package managers it is hard or impossible to . . .

- . . . install software from different distributions side-by-side

- . . . use multiple versions of the same software at the same time

- . . . automatically fetch and replace packages on demand

Motivation 1/2

With classical file systems and package managers it is hard or impossible to . . .

- . . . install software from different distributions side-by-side

- . . . use multiple versions of the same software at the same time

- . . . automatically fetch and replace packages on demand

Motivation 1/2

With classical file systems and package managers it is hard or impossible to . . .

- . . . install software from different distributions side-by-side

- . . . use multiple versions of the same software at the same time

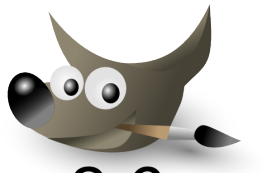
- . . . automatically fetch and replace packages on demand

Motivation 1/2

With classical file systems and package managers it is hard or impossible to . . .

- . . . install software from different distributions side-by-side

- . . . use multiple versions of the same software at the same time



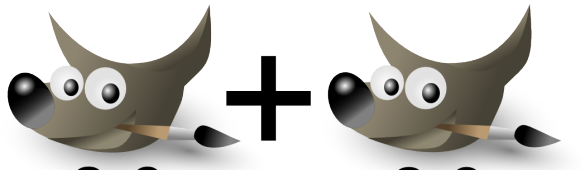
- . . . automatically fetch and replace packages on demand

Motivation 1/2

With classical file systems and package managers it is hard or impossible to . . .

- . . . install software from different distributions side-by-side

- . . . use multiple versions of the same software at the same time



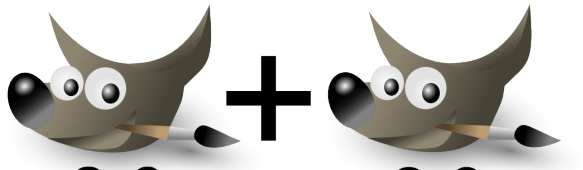
- . . . automatically fetch and replace packages on demand

Motivation 1/2

With classical file systems and package managers it is hard or impossible to . . .

- . . . install software from different distributions side-by-side

- . . . use multiple versions of the same software at the same time



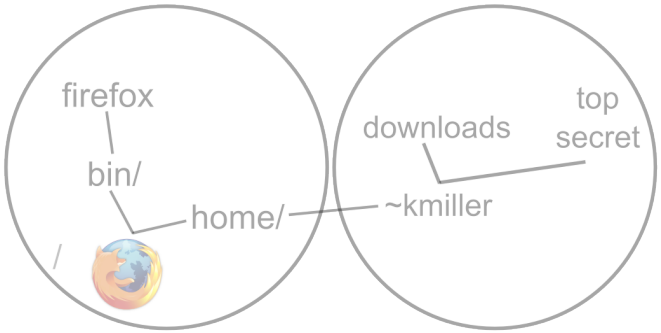
- . . . automatically fetch and replace packages on demand

Motivation 2/2

- . . . easily install any app without root privileges

```
$ sudo apt-get install mplayer
```

- . . . set different file access rights for *different apps* of the *same user*
  - Ever tried to jail users to their home for ssh sessions?
  - Allow Firefox to see only the “downloads” folder?



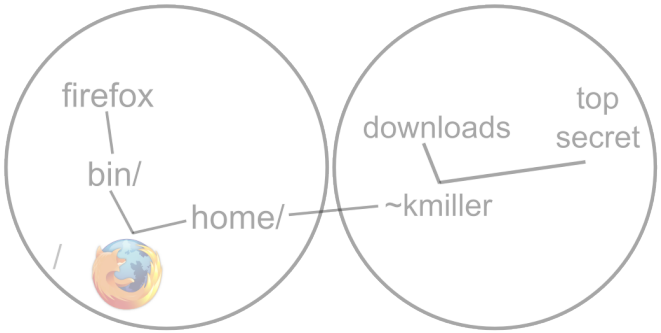
- These shortcomings all stem from
  - Naming conflicts
  - Imprecise specification of packages
  - Security/access rights issues

Motivation 2/2

- . . . easily install any app without root privileges

```
$ sudo apt-get install mplayer
```

- . . . set different file access rights for *different apps* of the *same user*
  - Ever tried to jail users to their home for ssh sessions?
  - Allow Firefox to see only the “downloads” folder?



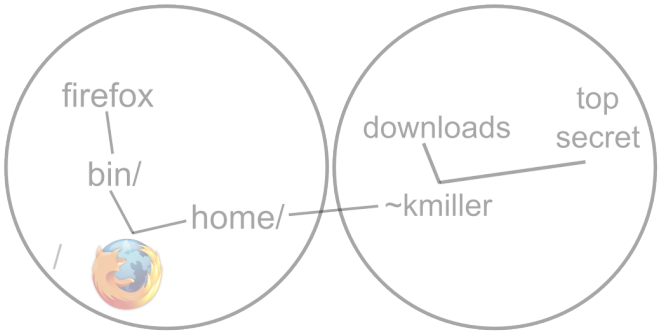
- These shortcomings all stem from
  - Naming conflicts
  - Imprecise specification of packages
  - Security/access rights issues

Motivation 2/2

- . . . easily install any app without root privileges

```
$ sudo apt-get install mplayer
```

- . . . set different file access rights for *different apps* of the *same user*
  - Ever tried to jail users to their home for ssh sessions?
  - Allow Firefox to see only the “downloads” folder?



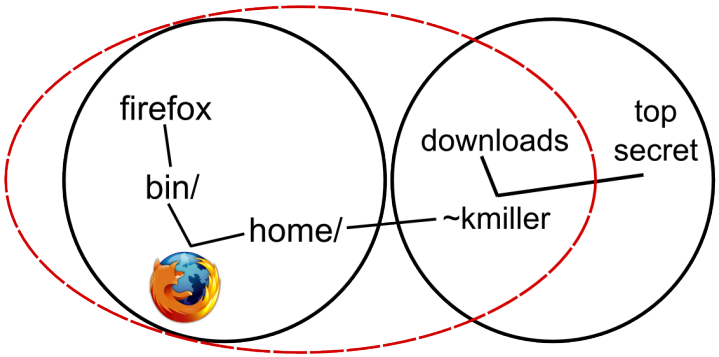
- These shortcomings all stem from
  - Naming conflicts
  - Imprecise specification of packages
  - Security/access rights issues

Motivation 2/2

- . . . easily install any app without root privileges

```
$ sudo apt-get install mplayer
```

- . . . set different file access rights for *different apps* of the *same user*
  - Ever tried to jail users to their home for ssh sessions?
  - Allow Firefox to see only the “downloads” folder?



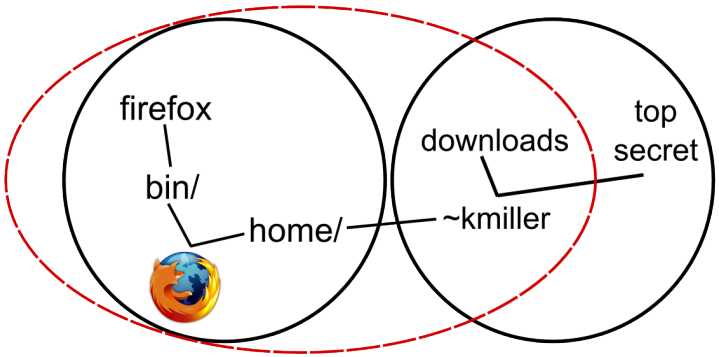
- These shortcomings all stem from
  - Naming conflicts
  - Imprecise specification of packages
  - Security/access rights issues

Motivation 2/2

- . . . easily install any app without root privileges

```
$ sudo apt-get install mplayer
```

- . . . set different file access rights for *different apps* of the *same user*
  - Ever tried to jail users to their home for ssh sessions?
  - Allow Firefox to see only the “downloads” folder?



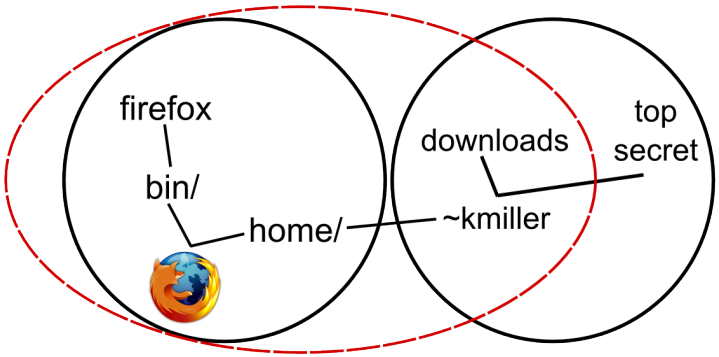
- These shortcomings all stem from
  - Naming conflicts
  - Imprecise specification of packages
  - Security/access rights issues

Motivation 2/2

- . . . easily install any app without root privileges

```
$ sudo apt-get install mplayer
```

- . . . set different file access rights for *different apps* of the *same user*
  - Ever tried to jail users to their home for ssh sessions?
  - Allow Firefox to see only the “downloads” folder?



- These shortcomings all stem from
  - Naming conflicts
  - Imprecise specification of packages
  - Security/access rights issues

Approach

A generic solution can be to break up the static, unified namespace thus creating **Dynamic File System Views!**

- Sandbox apps by giving every **(user, app)** tuple its own namespace
  - Made up of the application itself and its dependencies (e.g., shared objects)
  - Enhance privacy by making visibility of user content optional and explicit

- Separate handling of meta data from binaries and user content
  - Create view from meta data
  - Storage of objects and thus sharing of data stays intact
  - HDD is a cache for app data, but persistent storage for user content
  - The app and everything it needs is fetched and applied to namespace by installation in the application into the desktop

Approach

A generic solution can be to break up the static, unified namespace thus creating **Dynamic File System Views!**

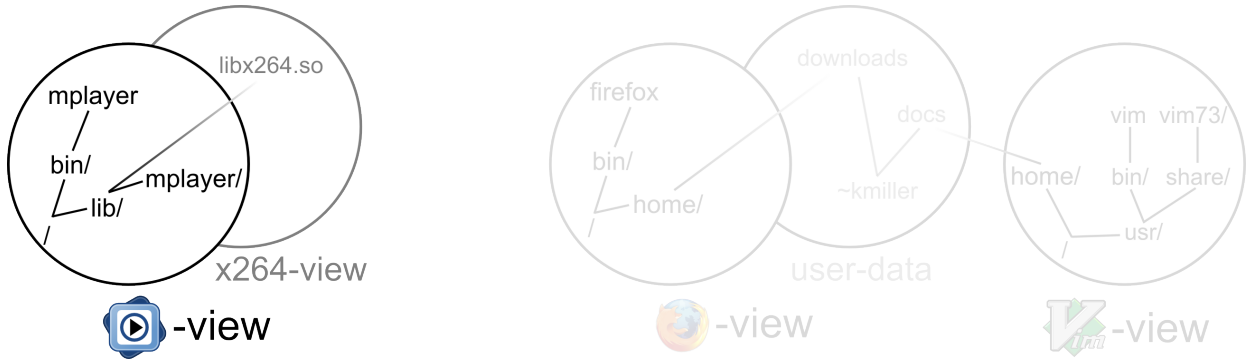
- Sandbox apps by giving every **(user, app)** tuple its own namespace
  - Made up of the application itself and its dependencies (e.g., shared objects)
  - Enhance privacy by making visibility of user content optional and explicit

- Separate handling of meta data from binaries and user content
  - Create view from meta data
  - Storage of objects and thus sharing of data stays intact
  - HDD is a cache for app data, but persistent storage for user content
    - The app and everything the app needs is fetched and cached transparently
    - Installation is integrating the application into the desktop

Approach

A generic solution can be to break up the static, unified namespace thus creating **Dynamic File System Views!**

- Sandbox apps by giving every **(user, app)** tuple its own namespace
  - Made up of the application itself and its dependencies (e.g., shared objects)
  - Enhance privacy by making visibility of user content optional and explicit

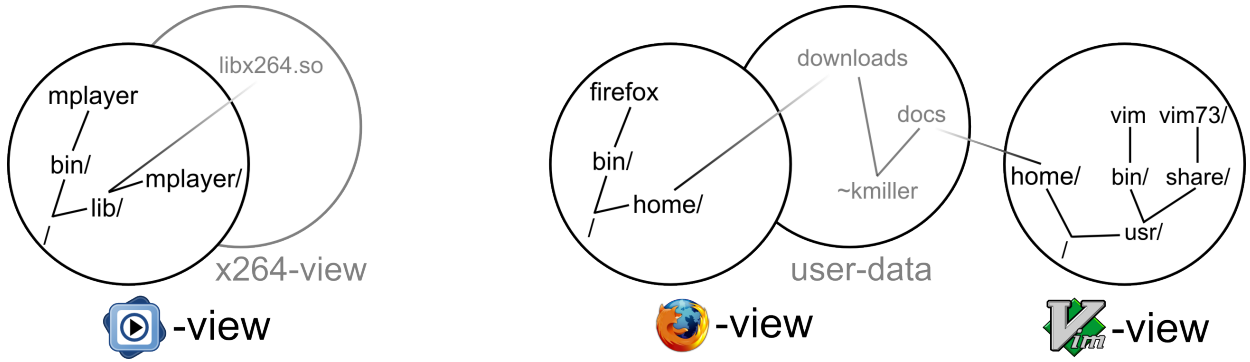


- Separate handling of meta data from binaries and user content
  - Create view from meta data
  - Storage of objects and thus sharing of data stays intact
  - HDD is a cache for app data, but persistent storage for user content
  - The app and everything it needs is fetched and applied for example by installation to immediately run the application from the desktop

Approach

A generic solution can be to break up the static, unified namespace thus creating **Dynamic File System Views!**

- Sandbox apps by giving every **(user, app)** tuple its own namespace
  - Made up of the application itself and its dependencies (e.g., shared objects)
  - Enhance privacy by making visibility of user content optional and explicit

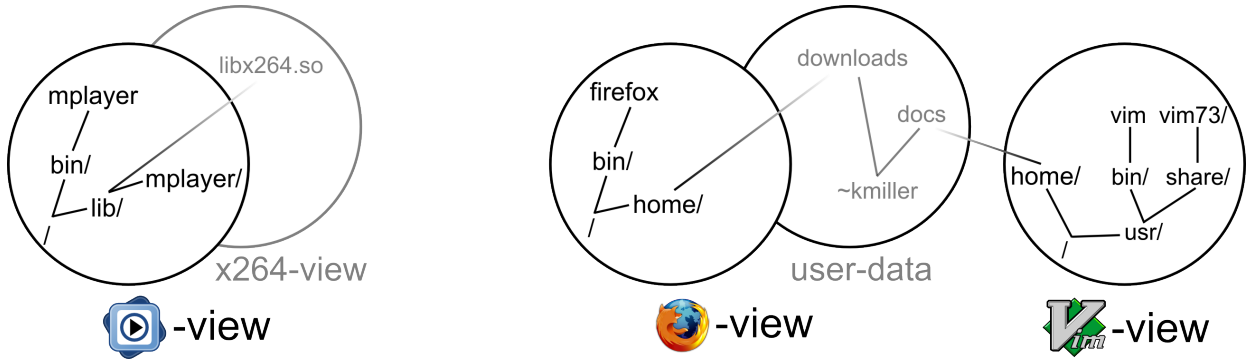


- Separate handling of meta data from binaries and user content
  - Create view from meta data
  - Storage of objects and thus sharing of data stays intact
  - HDD is a cache for app data, but persistent storage for user content
  - The app and everything it needs is fetched and cached transparently
  - Installation is integrating the application into the user's view

Approach

A generic solution can be to break up the static, unified namespace thus creating **Dynamic File System Views!**

- Sandbox apps by giving every **(user, app)** tuple its own namespace
  - Made up of the application itself and its dependencies (e.g., shared objects)
  - Enhance privacy by making visibility of user content optional and explicit

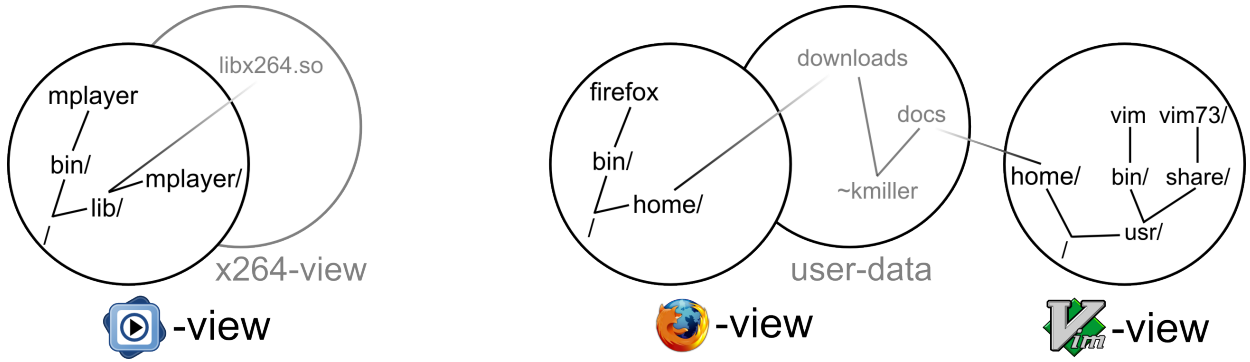


- Separate handling of meta data from binaries and user content
  - Create view from meta data
  - Storage of objects and thus sharing of data stays intact
  - HDD is a cache for app data, but persistent storage for user content
    - The app and everything the app needs is fetched and cached transparently
    - Installation is integrating the application into the desktop

Approach

A generic solution can be to break up the static, unified namespace thus creating **Dynamic File System Views!**

- Sandbox apps by giving every **(user, app)** tuple its own namespace
  - Made up of the application itself and its dependencies (e.g., shared objects)
  - Enhance privacy by making visibility of user content optional and explicit

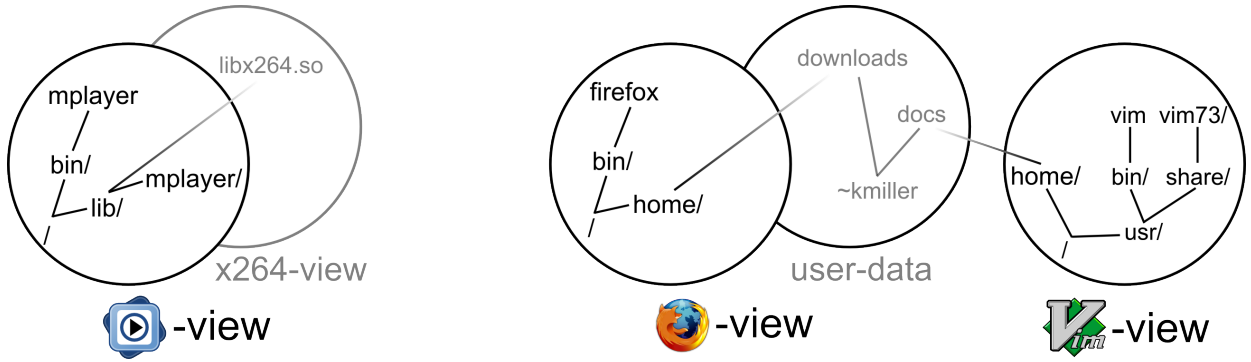


- Separate handling of meta data from binaries and user content
  - Create view from meta data
  - Storage of objects and thus sharing of data stays intact
  - HDD is a cache for app data, but persistent storage for user content
    - The app and everything the app needs is fetched and cached transparently
    - Installation is integrating the application into the desktop

Approach

A generic solution can be to break up the static, unified namespace thus creating **Dynamic File System Views!**

- Sandbox apps by giving every **(user, app)** tuple its own namespace
  - Made up of the application itself and its dependencies (e.g., shared objects)
  - Enhance privacy by making visibility of user content optional and explicit

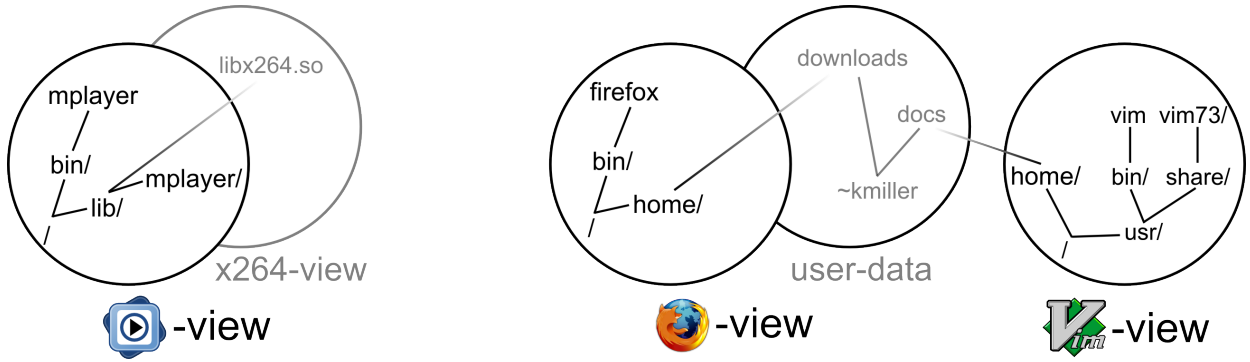


- Separate handling of meta data from binaries and user content
  - Create view from meta data
  - Storage of objects and thus sharing of data stays intact
  - HDD is a cache for app data, but persistent storage for user content
    - The app and everything the app needs is fetched and cached transparently
    - Installation is integrating the application into the desktop

Approach

A generic solution can be to break up the static, unified namespace thus creating **Dynamic File System Views!**

- Sandbox apps by giving every **(user, app)** tuple its own namespace
  - Made up of the application itself and its dependencies (e.g., shared objects)
  - Enhance privacy by making visibility of user content optional and explicit



- Separate handling of meta data from binaries and user content
  - Create view from meta data
  - Storage of objects and thus sharing of data stays intact
  - HDD is a cache for app data, but persistent storage for user content
    - The app and everything the app needs is fetched and cached transparently
    - Installation is integrating the application into the desktop

Challenges

- How does desktop integration work?
  - How do you start apps?
  - How do you share data between apps and users?
  - How do apps interplay?
    - What happens if you click a `mailto:` link in a browser?
- How do you find the min. dependency set?
  - RPM's dependency list is incomplete and based on names
- How high is the toll you need to pay?
  - Runtime, storage, bandwidth overhead?

- Let's talk about it
  - There are plenty of design options to enhance the state of the art (0-install, packaging concepts, chroot, compartments, virtualization, ...)
  - I am looking forward to hearing your comments
  - Come to my poster for implementation ideas and discussions

Challenges

- How does desktop integration work?
  - How do you start apps?
  - How do you share data between apps and users?
  - How do apps interplay?
    - What happens if you click a `mailto:` link in a browser?
- How do you find the min. dependency set?
  - RPM's dependency list is incomplete and based on names
- How high is the toll you need to pay?
  - Runtime, storage, bandwidth overhead?

- Let's talk about it
  - There are plenty of design options to enhance the state of the art (0-install, packaging concepts, chroot, compartments, virtualization, ...)
  - I am looking forward to hearing your comments
  - Come to my poster for implementation ideas and discussions

Challenges

- How does desktop integration work?
  - How do you start apps?
  - How do you share data between apps and users?
  - How do apps interplay?
    - What happens if you click a `mailto:` link in a browser?
- How do you find the min. dependency set?
  - RPM's dependency list is incomplete and based on names
- How high is the toll you need to pay?
  - Runtime, storage, bandwidth overhead?

- Let's talk about it
  - There are plenty of design options to enhance the state of the art (0-install, packaging concepts, chroot, compartments, virtualization, . . .)
  - I am looking forward to hearing your comments
  - Come to my poster for implementation ideas and discussions

Challenges

- How does desktop integration work?
  - How do you start apps?
  - How do you share data between apps and users?
  - How do apps interplay?
    - What happens if you click a `mailto:` link in a browser?
- How do you find the min. dependency set?
  - RPM's dependency list is incomplete and based on names
- How high is the toll you need to pay?
  - Runtime, storage, bandwidth overhead?

- Let's talk about it
  - There are plenty of design options to enhance the state of the art (0-install, packaging concepts, chroot, compartments, virtualization, . . .)
  - I am looking forward to hearing your comments
  - Come to my poster for implementation ideas and discussions