Karlsruhe Institute of Technology
System Architecture Group
http://os.ibds.kit.edu

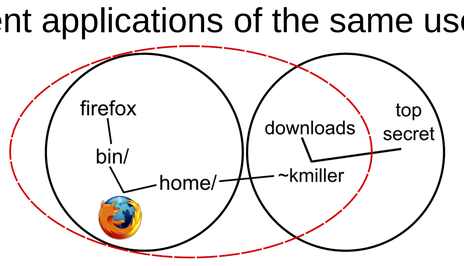
A case for dynamic file system views

Konrad Miller <miller@kit.edu>

1. Motivation

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

- Set different access rights for different applications of the same user
  - Ever tried to jail users to their home for ssh sessions?
  - Allow Firefox to only see the “Downloads” folder?



- Install software from different distributions or multiple versions of the same software side-by-side

- Install software without administrator privileges
- Automatically fetch and replace packages on demand

~~sudo~~ apt-get install x

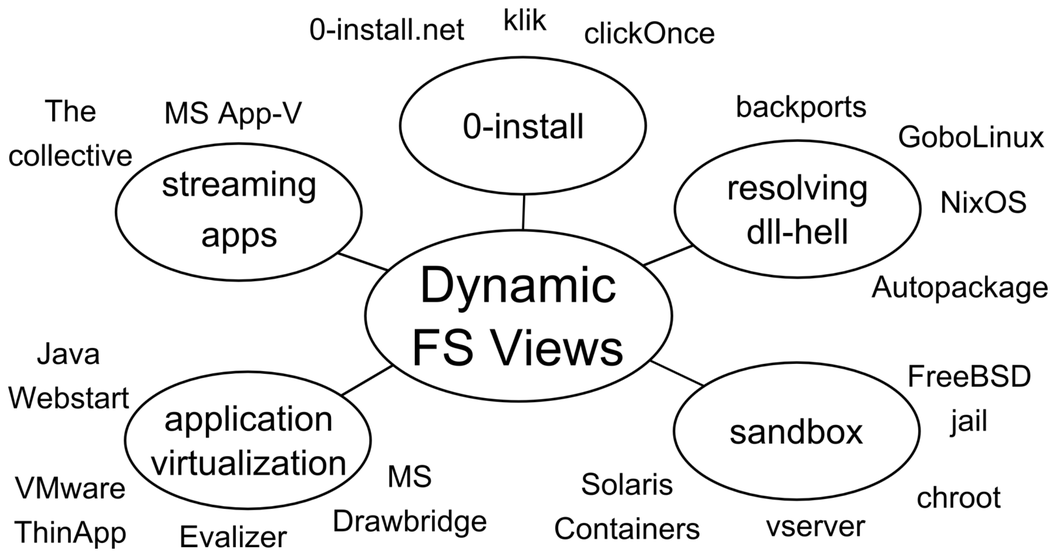
- Keep software up to date at all times and update across distribution versions
- Distribute and re-distribute files across device boundaries
- Have a personalized software selection installed across multiple machines and systems (“software mobility”)

2. Vision

- Automatically create minimal sandboxes for all applications
  - Avoid naming conflicts rather than resolving them
  - Allow different versions of files to coexist for different apps
  - Maintain maximal reuse of components among applications
- Make sharing of user content optional and explicit
  - Every application only sees what it needs
  - Show user content on demand only
  - “If you can’t name it, you can’t touch it”

3. Existing approaches

- Virtualization of entire operating systems or single applications
- Sandboxes, chroot, jail, vserver
- Zero-install systems, portable apps, backports
- Purely functional and traditional package managers

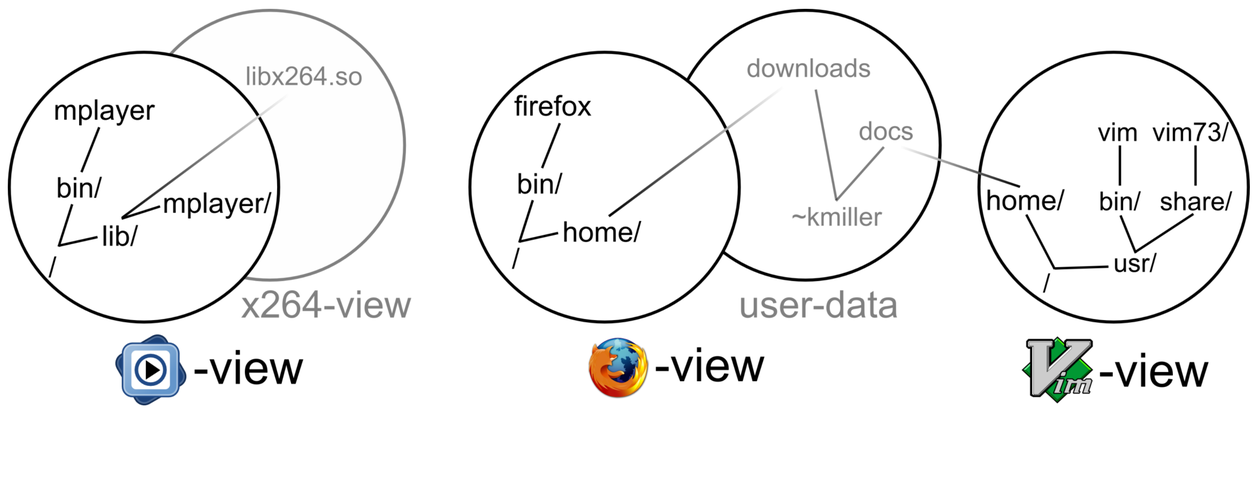


4. Shortcomings of existing approaches

- The existing approaches only touch on the problems at hand or fight the symptoms but not the causes
- Many open questions remain
  - Desktop integration
  - Sharing across domain boundaries (e.g., among VMs)
  - De-duplication of components in memory and on disk
  - Automation
  - Reuse of existing systems

5. Approach

- Break up the unified file system namespace
  - Let every (user, app) tuple have their own namespace
  - Satisfy dependencies by making them visible to the respective application’s namespace in the expected place
- Make sharing and desktop integration explicit
  - Reduce installation and deinstallation to hooking into or unhooking from the desktop integration
  - Govern access to user content
- Clearly separate binaries, configuration, and user content
  - Make dependencies data specific, not nominal
  - Use local storage as a cache for application data
    - Application data is fetched and cached on demand (e.g., on first run or when integrated into desktop)
    - Old data will eventually be replaced by new data (i.e., old applications will fade away as new apps/new versions are used)
  - Store user content persistently
- Create meta-namespace for browsing application repositories and starting applications



6. Challenges

- How usable and intuitive is the approach?
- How can you find the minimal dependency set?
- How do you share data between applications and users?
- How fast/efficient/scalable is the approach?
- How much additional memory does the approach use?
- What security implications does the approach introduce?
- How can software mobility be implemented?