Lecture 12

RFS – A Network File System for Mobile Devices and the Cloud

Yuan Dong, Jinzhan Peng, Dawei Wang, Haiyang Zhu, Fang Wang, Sun C. Chan, Michael P. Mesnier

Operating Systems Practical

18 December, 2013

SOA paper crunch

Outline

Context

- Mobile Devices
  - increasing number of applications - lots of data - key issue: access to dependable storage
  - cloud storage - attractive
- Cloud storage for mobile devices - new issues
  - unpredictable wireless network connectivity
  - data privacy over the network
- Solution: RaindropFS
  - *'wireless friendly'*
  - *'network file-system'*
  - *'... for mobile devices and the cloud'*

- Wireless friendly
  - network connectivity
  - average bandwidth around the world (2010 study): 105 Kbps. . . 7.2 MBps
- Network file system - issues to be resolved
  - unpredictable connectivity of the wireless network
  - data privacy introduced by the cloud

Issue #1: Unpredictable Wireless Connectivity

- local storage cache - the right approach
- the *'unpredictable connectivity'* problem
- server-centric management
  - server controlled synchronization between server and client caches
  - not well suited for unpredictable wireless connectivity
  - constrains such as varying bandwidth cost, battery life etc.
- solution: management controlled by the client
  - continuous file consistency not a goal

Issue #2: Data Privacy

- mobile devices - single user
- public clouds: no satisfactory solution to privacy
- solution: client-centric approach which depends on applications

Contributions - Raindrop FS

- device-aware cache management
  - different networks
  - different costs
  - energy-awareness
- different levels of client-driven data security and privacy policies
- client-aware optimizations at the server
  - previous usage recorded
  - predict usage
  - server *prepush*
- partial file caching for large files
- integrate with Amazon S3 cloud storage
- tested Android running on top of Raindrop FS (x86-based netbook)

Outline

Design Goals

- Cloud storage abstraction
  - ability to connect using any interface to cloud storage
  - synchronize data efficiently and transparently
- Connection optimization
  - adapt connections based on the mobile device's connectivity
- Client data protection
  - client-controlled user data security and privacy

Cloud storage abstraction

- Components
  - RaindropFS server responsible for mapping files to the cloud
  - RaindropFS client residing on the mobile devices
  - HTTP for client-server communication
- Features
  - Cloud cache (64 MB segment = unit of transmission between RFS server and cloud provider)
  - Cloud adapter - support for multiple clouds

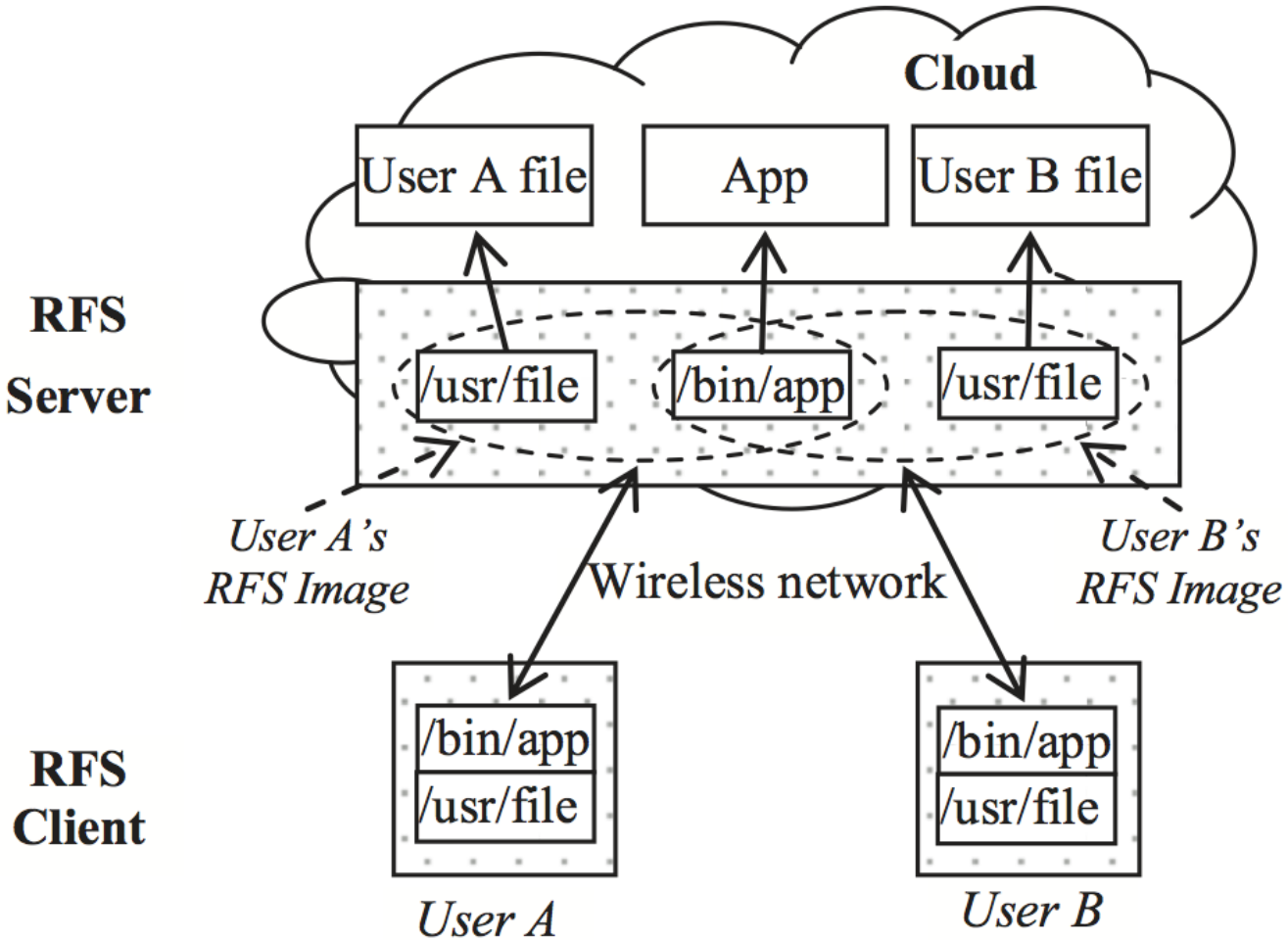

Figura: The Raindrop FS client-server model

Network communication in Raindrop FS

- Demand fetching
  - update local storage from cloud
- Synchronization
  - transfer data between client-cache and cloud
- Consistency maintenance
  - consistency of files over multiple clients

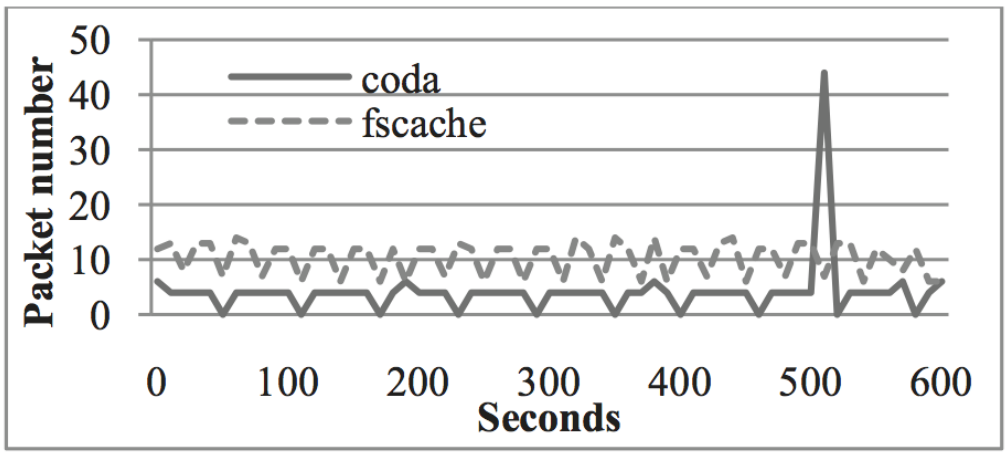

Figura: Packets transmitted over WiFi for sync/consistency maintenance

Connection optimization

- Network file-system design
  - server-centric - server controls synchronization and consistency maintenance
  - client-centric - clients decide when to synchronize
- Consistency models
  - close-to-open - each *close()* requires interaction with the server
  - RFS's client-controlled consistency - synchronization at client-specified sync points; server maintains version numbers for files and folders

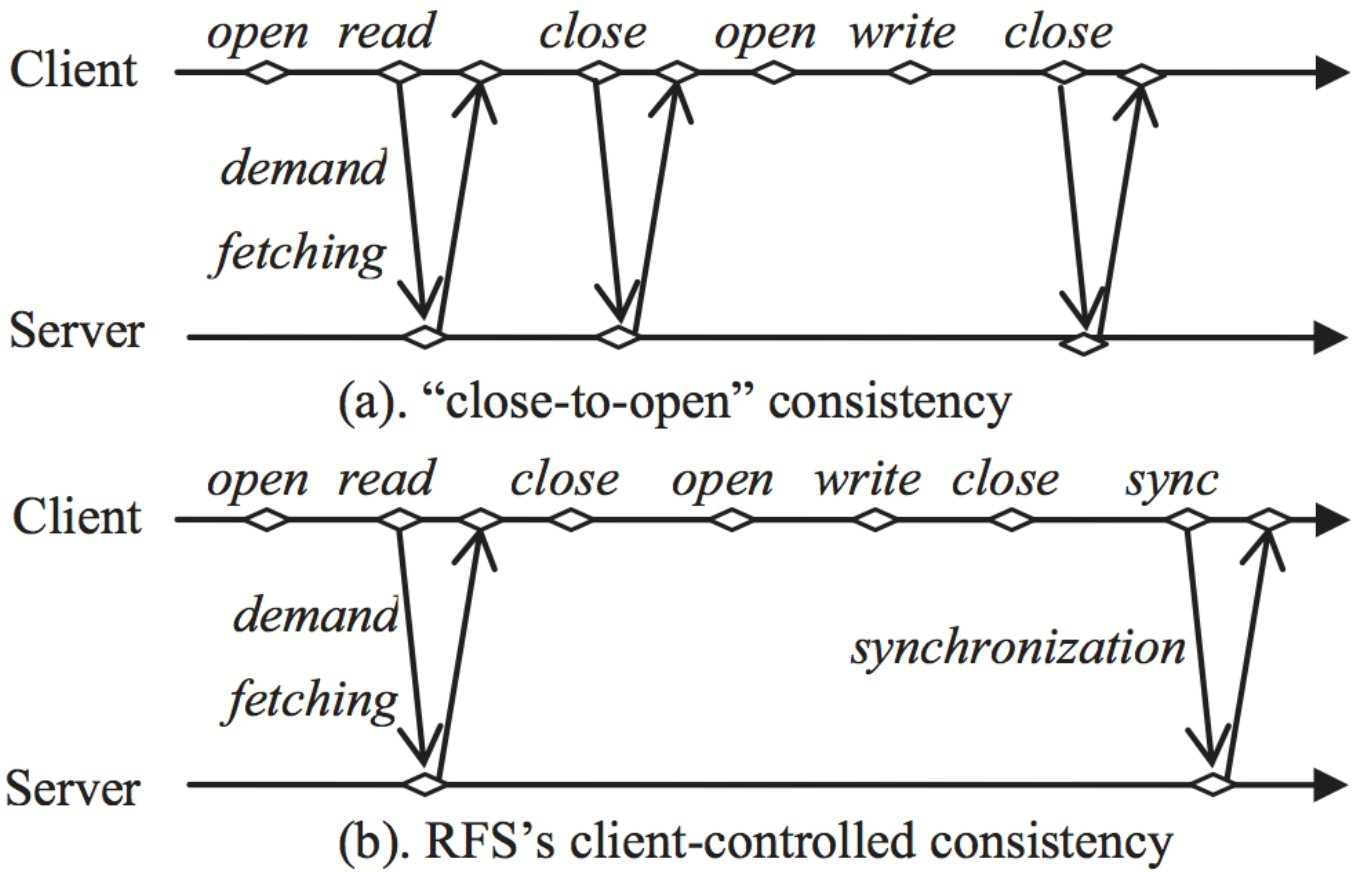

Figura: Consistency models

Device State	Description
Network cost	no signal, free, low, high
Battery capacity	low, high, unlimited
CPU load	low, medium, high
Memory load	low, medium, high

Figura: Device states used in RFS

- multiple levels per state
- user configuration is required to define sync conditions - example: low-cost/free network, high battery capacity, low CPU and memory load

- logging all file operations: costly
- solution: status vector
  (files: 5 bits; folders: 4 bits - no `modify` bit)
  `<deleted, created, renamed, attr-changed, modified>`

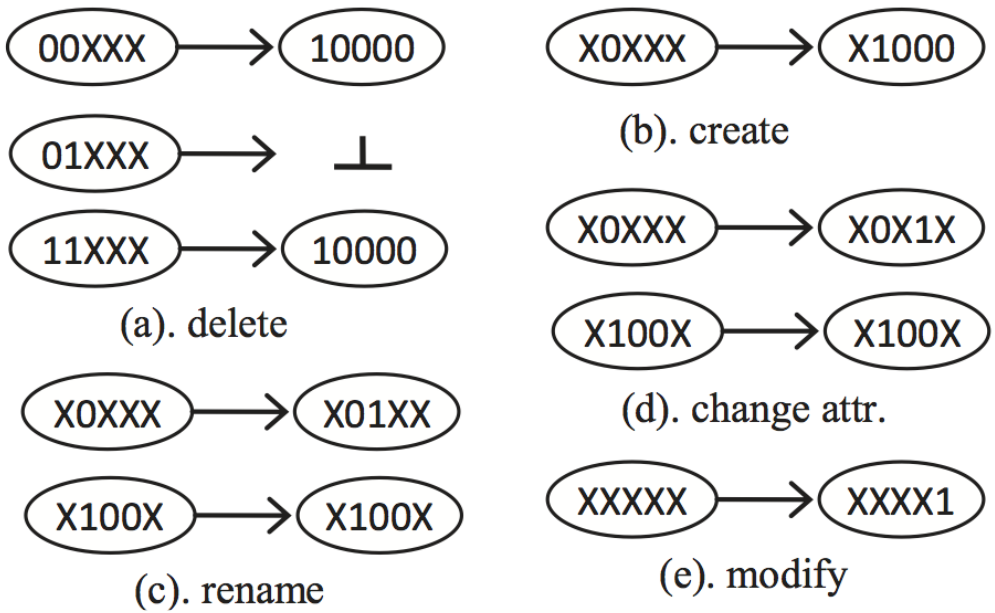

Figura: File status vector transition

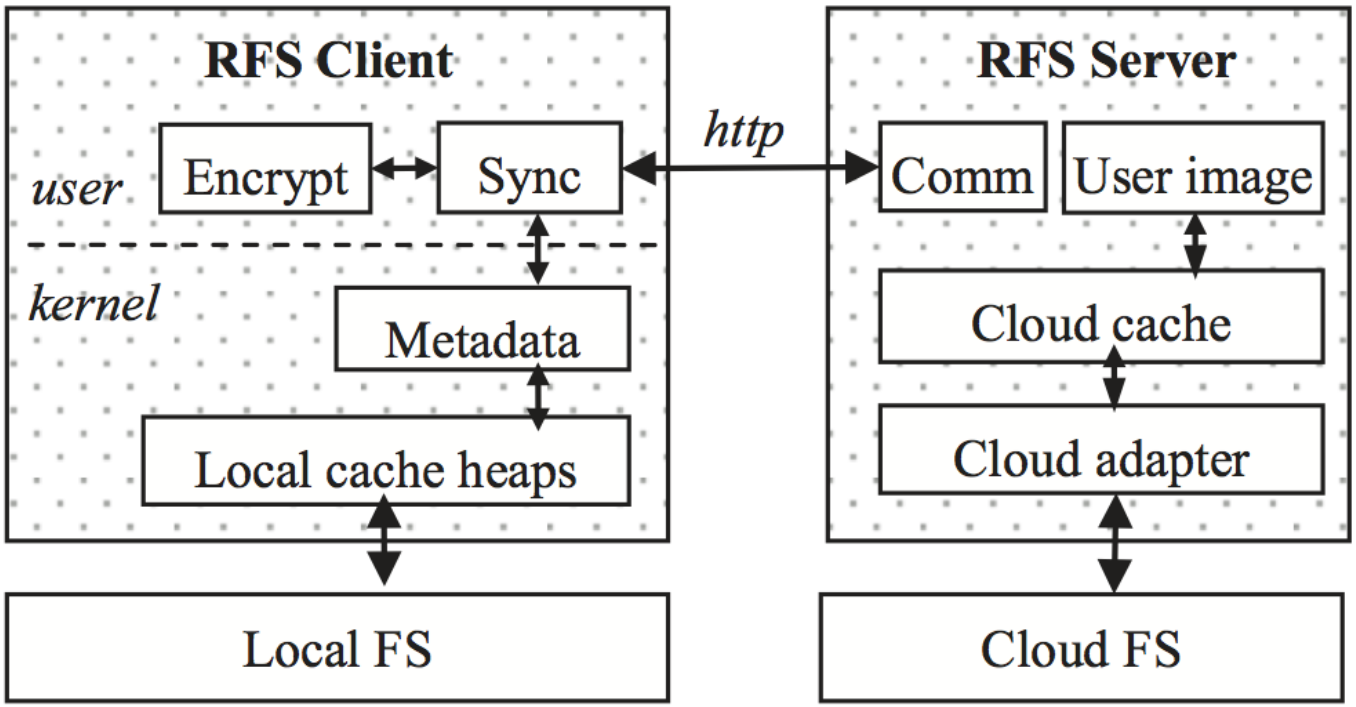

Figura: RFS Architecture

Outline

Overview

- Raindrop FS client
  - Linux kernel module for performance critical components: local cache heaps, metadata manager
  - user space: Sync daemon, encryption engine
- Raindrop FS server
  - built on cloud storage directly
  - RFS files: regular files, links to public Internet resources
  - uses HTTP 1.1 and SOAP, allowing clients to bypass network firewalls, use gateways or proxies
  - *zlib* compression algorithm for communication efficiency

Managing local cache heaps

- other network file systems: local file mapping, separate partition
- in RFS: partial file caching
- all cached blocks for a given RFS file
  - stored in one file
  - organized as a heap
- heaps are preallocated (blocks physically contiguous)
- recycle action (cache evictions) for freeing up space (heap can no longer accommodate new cached blocks)
- support for block-pinning

Managing local cache heaps

- on mount: metadata loaded into memory
- client maintains a dirty list of modified files and directories
- bitmap per file indicating dirty blocks that need sync
- if metadata updates from server: client invalidates cached blocks

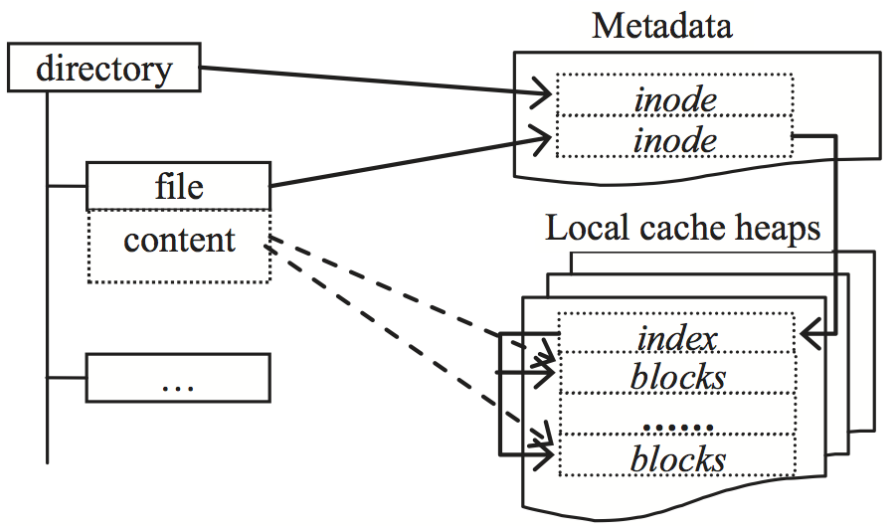

Figura: How RFS directories & files map to a local cache

Integrating the RFS server and the cloud

- *metadata* in MySQL (adapts to Amazon's RDS)
- *file data* on Amazon S3
  (thin storage API with Get/Put/Delete)
- updates pushed to the server in 64 MB segments
  (communication efficiency)
- RFS RESTful web service API (without management options)
  - (files, directories) create, delete, rename
  - (files, directories) set/get entity attributes
  - (files) read, write file blocks
  - (directories) read - retrieve file list
  - update - retrieve update information from the server within specified time interval

Data privacy

- client-side encryption
- user-policy based (not everything needs to be encrypted)
- AES used on the client for read/write operations
- metadata currently not encrypted
  - reveals structure of the client file system to the server
  - tradeoff that allows block-level optimizations such as updates of the system files

Server prepush

- used to speed up cold file fetching (unknown files)
- file's block access pattern *collected across multiple users*
- more blocks are pushed on a new request from a client based on the access pattern
- access patterns stored per file in groups of tuples `<start-block, end-block>`
- requests trigger lookups into the groups of tuples
  - on match, server prepushes blocks
  - on miss, server records a new access pattern
- groups can be merged if adjacent
- *prepush hint* added to client requests

Outline

Conditions

- Mobile device
  (Asus EeePC 1000H, 1.6 GHz Atom CPU, 1GB RAM)
- Ubuntu Mobile
- Compared to FScache (with NFS as server) and Coda
- Connections
  - Wired (100 MBps Ethernet)
  - WiFi (54 MBps over a D-Link wireless router)
  - WCDMA (7.2 MBps 3G network)
- Benchmarks
  - grep - search for a text over the mounted network FS
  - copyin - copy from the local FS to the mounted network FS
  - copyout - copy from mounted network FS to the local FS
  - install - untar archive in the network FS
- Cold vs. Warm (data in local cache) tests

- Warm mode testing - RFS lazy sync design
- FScache only caches for reading - no changes for warm mode copyin & install tests

Mode	Network	FS	Grep	Copyin	Copyout	Install
Cold	Wired	*FScache*	*11.63*	*80.61*	*11.58*	*49.83*
		Coda	*34.65*	*13.16*	*39.26*	*17.03*
		RFS	***10.58***	***6.01***	***15.59***	***12.21***
	WiFi	*FScache*	*41.65*	*130.91*	*47.78*	*168.22*
		Coda	*63.95*	*13.40*	*63.98*	*32.00*
		RFS	***50.50***	***6.92***	***49.49***	***35.44***
	WCDMA	***RFS***	***2710***	***7.01***	***4217***	***1693***
Warm	Wired	*FScache*	*1.23*	*74.99*	*1.78*	*48.89*
		Coda	*0.65*	*5.42*	*0.76*	*7.92*
		RFS	***0.27***	***2.05***	***1.01***	***4.35***
	WiFi	*FScache*	*3.27*	*105.71*	*4.26*	*145.28*
		Coda	*0.58*	*5.69*	*0.86*	*8.04*
		RFS	***0.28***	***1.82***	***1.23***	***4.40***
	WCDMA	***RFS***	***0.41***	***2.01***	***1.20***	***4.01***

Figura: Benchmark performance (seconds)

- Coda - small number of request for sync (network type has no impact)
- RFS - no requests sent

Mode	Network	FS	Grep	Copyin	Copyout	Install
Cold	Wired	*FScache*	*102k*	*46.9k*	*100k*	*105k*
		Coda	*111k*	*612*	*111k*	*53.5k*
		RFS	***90k***	***12***	***107k***	***49.8k***
	WiFi	*FScache*	*106k*	*67.1k*	*106k*	*158k*
		Coda	*116k*	*543*	*111k*	*54.5k*
		RFS	***90.7k***	***12***	***107k***	***49.9k***
	WCDMA	***RFS***	***145K***	***81***	***118k***	***54.8k***
Warm	Wired	*FScache*	*5.1k*	*46.8k*	*4.9k*	*57k*
		Coda	*10*	*485*	*10*	*2.1k*
		RFS	***0***	***0***	***0***	***0***
	WiFi	*FScache*	*5.2k*	*67.1k*	*4.7k*	*112k*
		Coda	*10*	*489*	*10*	*3.3k*
		RFS	***0***	***0***	***0***	***0***
	WCDMA	***RFS***	***0***	***0***	***0***	***0***

Figura: Benchmark packets

Privacy overhead

- encryption in read/write synchronization operations
- no encryption in local operations
- lower overhead on low bandwidth connections

Type	Size (MB)	No-enc (sec)	Encrypt (sec)	Diff (sec)	Overh ead
*Local	605	8	29	21	263%
*Wired	605	54	71	17	31%
Wired	605	56	104	48	85.7%
WiFi	605	193	234	41	21.2%
WCDMA	12	101	104	3	3.0%

Figura: Privacy overhead of the read operation

Case Study Description

- Android-x86 platform on EeePC 1000H
- ported RFS to the Android Linux kernel
- changed /root/init.rc to point to the Android system on the RFS server
- only Linux kernel & ramdisk on local FS
- boot Android over RFS
- Android system files are public files (not encrypted)

- Android system: 1322 files, 463 MB total
- RFS on boot transfer: 167 files, 49.4 MB

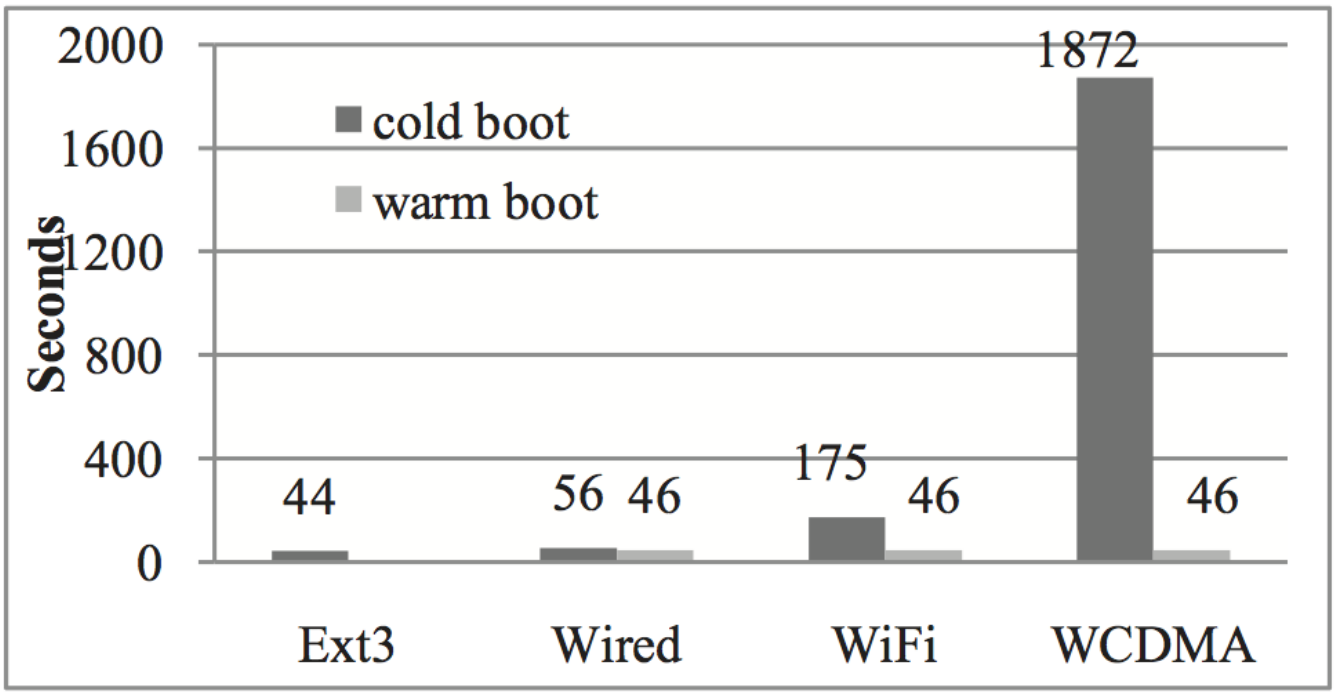

Figura: Booting Android over RFS

Filename	Size (MB)	Access blocks	Percent
libopencore_player.so	12.4	220	7%
libopencore_common.so	9.4	233	10%
libopencore_net_support.so	6.8	121	7%
libskia.so	6.8	332	19%
libandroid_runtime.so	5.6	182	13%
libicui18n.so	4.8	260	21%
/libdvm.so	4.5	203	18%
libopencore_author.so	4.4	108	10%
framework-res.apk	4.2	1044	97%
libicuuc.so	3.6	279	30%
libcrypto.so	3.3	259	30%
DroidSansFallback.ttf	2.9	598	79%

Figura: Top 12 files in Android booting

Android booting with server prepush

- mmap() to load executables
- each page fault triggers remote file block read
- with server prepush
  - request count drops 24x
  - transmission size drops by 10%

Server prepsuh	Request number	Total size (MB)	Xmt size (MB)
Off	12350	49.4	28.4
On	509	49.8	25.9

Figura: Server prepush on Android booting

Android booting with server prepush (speedup)

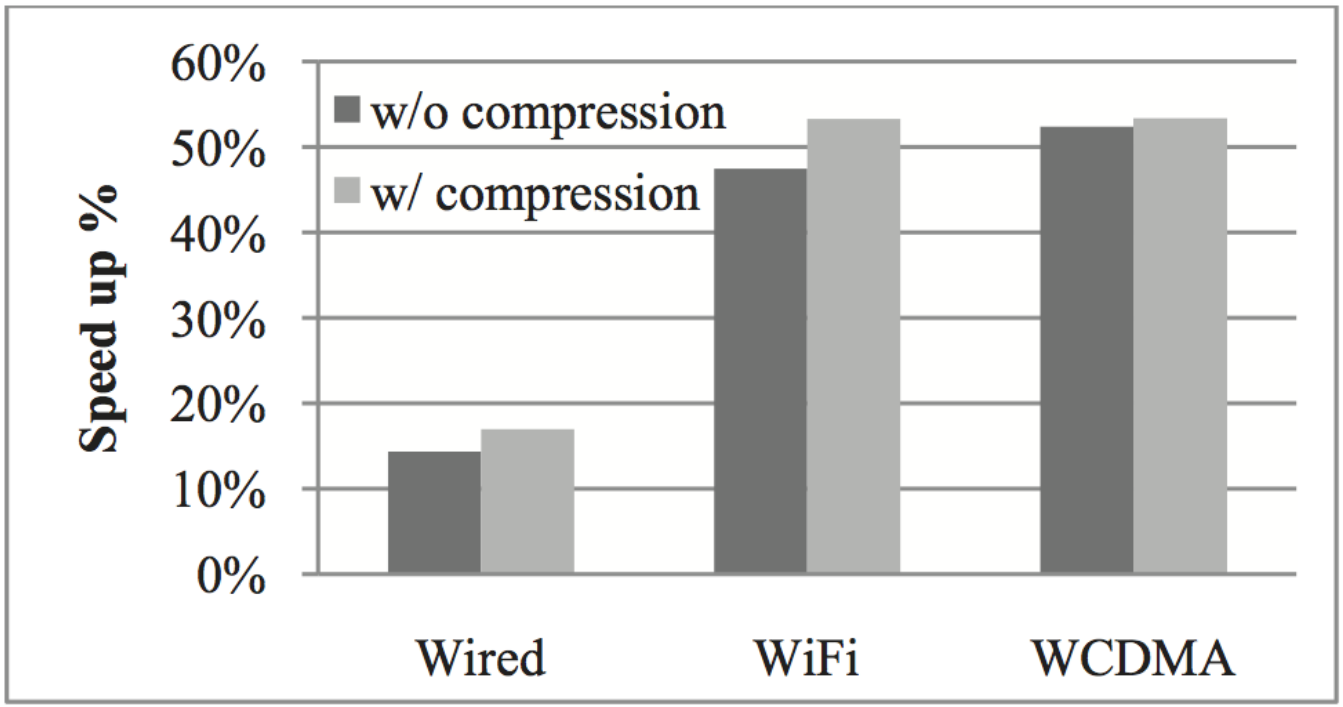

Figura: Booting Android with server prepush

Outline

Related work (1)

- Cache management
  - previous work - mainly server-centric
  - popular network file systems not designed for mobile
  - RFS - clients control
- Disconnected opperation
  - Local cache: Coda, Version Control systems (CVS, SVN), browsers (offline mode), HTML5 (Local Storage)
  - RFS - client-controlled caching
- File Systems for weak connectivity - Moxie
  - reduced data transfer, addresses weak connections
  - protects data in transmission (data privacy) vs. RFS (information privacy)
  - no prefetching

Related work (2)

- Prefetch vs. Prepush
  - lots of literature on prefetching
  - MFS uses prefetching for adapting data access patterns to network availability
  - server prepush - block access patterns collected across multiple users
- Privacy for the cloud
  - NFS, Coda rely on trusted administrators, taking into account only access controls
  - Amazon/Rackspace provide cloud service hosting but with no data privacy
- Applications atop of cloud storage
  - Dropbox
  - Jungle Disk, S3 Backup - client side encryption
  - usually targeted for personal backup systems

Comparison between RFS and other mobile file systems

Mobile FSes	Partial caching	Consistency	Prefetching	Write caching	Weak connect	Mobile target	Network aware	Battery aware	Privacy	Transport protocol
AFS	*No*	*Close-to-open*	*No*	*Yes*	*No*	*No*	*No*	*No*	*No*	*RPC*
Coda	*No*	*Callback*	*No*	*Yes*	*Yes*	*Yes*	*Yes*	*No*	*Yes*	*Multi RPC*
FScache	*Yes*	*Traditional*	*No*	*No*	*No*	*No*	*No*	*No*	*No*	*RPC*
LBFS	*No*	*Close-to-open*	*No*	*No*	*Yes*	*No*	*No*	*No*	*No*	*NFS-like RPC*
MFS	*No*	*Close-to-open*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*No*	*No*	*Adaptive RPC*
Moxie	*Yes*	*Close-to-open*	*No*	*Yes*	*Yes*	*Yes*	*Yes*	*No*	*Yes*	*Custom RPC*
RFS	*Yes*	*Client-control*	*Server push*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*	*HTTP*

Figura: Comparison between RFS and other mobile file systems

Outline

Conclusion

- client-centric design addresses
  - unpredictable wireless network connectivity
  - data privacy over the network
- client-control synchronization and consistency
- new optimizations
  - *server prepush* for speeding up cold file fetching
  - *reintegration of changes* in the device's with the cloud
- future work
  - user-specified management policies
  - hashing file contents for detecting redundant blocks
  - additional encryption algorithms (e.g. public-key)

Outline

Keywords

- mobile
- network file system
- cloud storage
- data privacy
- device-aware cache management
- network aware
- battery aware
- partial file caching
- data synchronization
- server prepush
- Android
- Amazon S3

Resources

- Yuan Dong, Haiyang Zhu, Jinzhan Peng, Fang Wang, Michael P. Mesnier, Dawei Wang, and Sun C. Chan. – **RFS: a network file system for mobile devices and the cloud**, *SIGOPS Operating Systems Review*, Volume 45, Issue 1 (February 2011), 101-111. `http://doi.acm.org/10.1145/1945023.1945036`