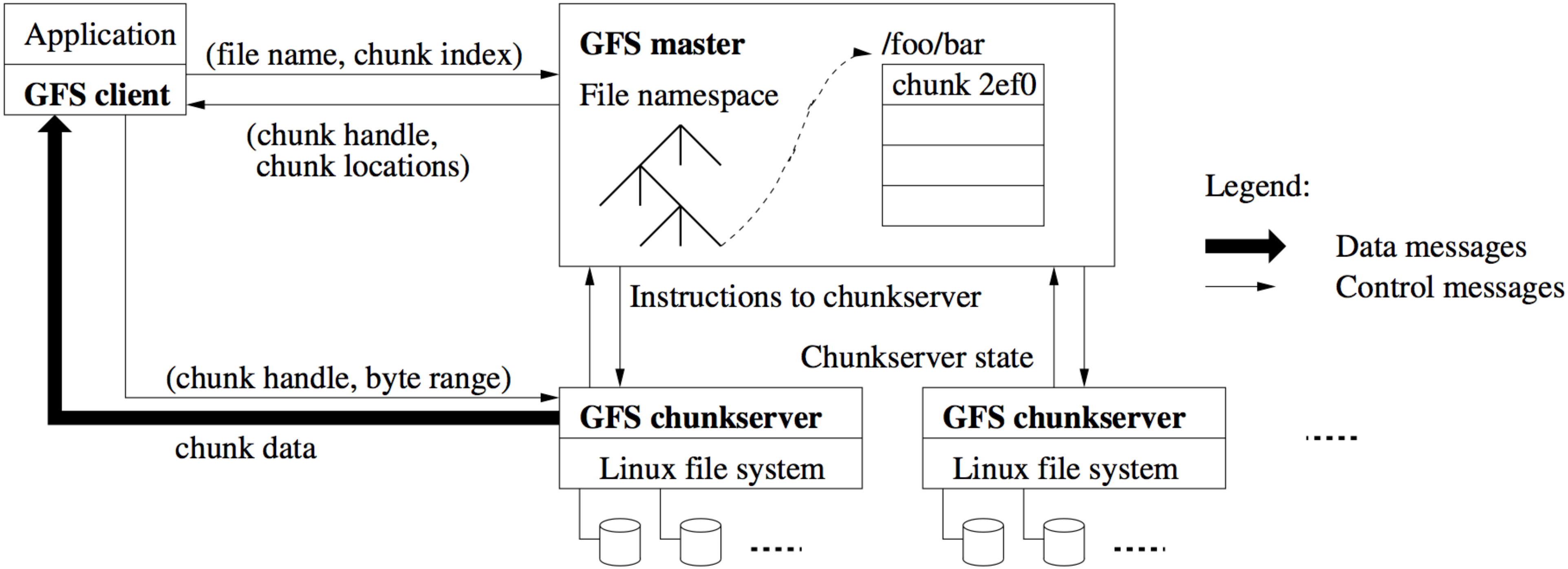
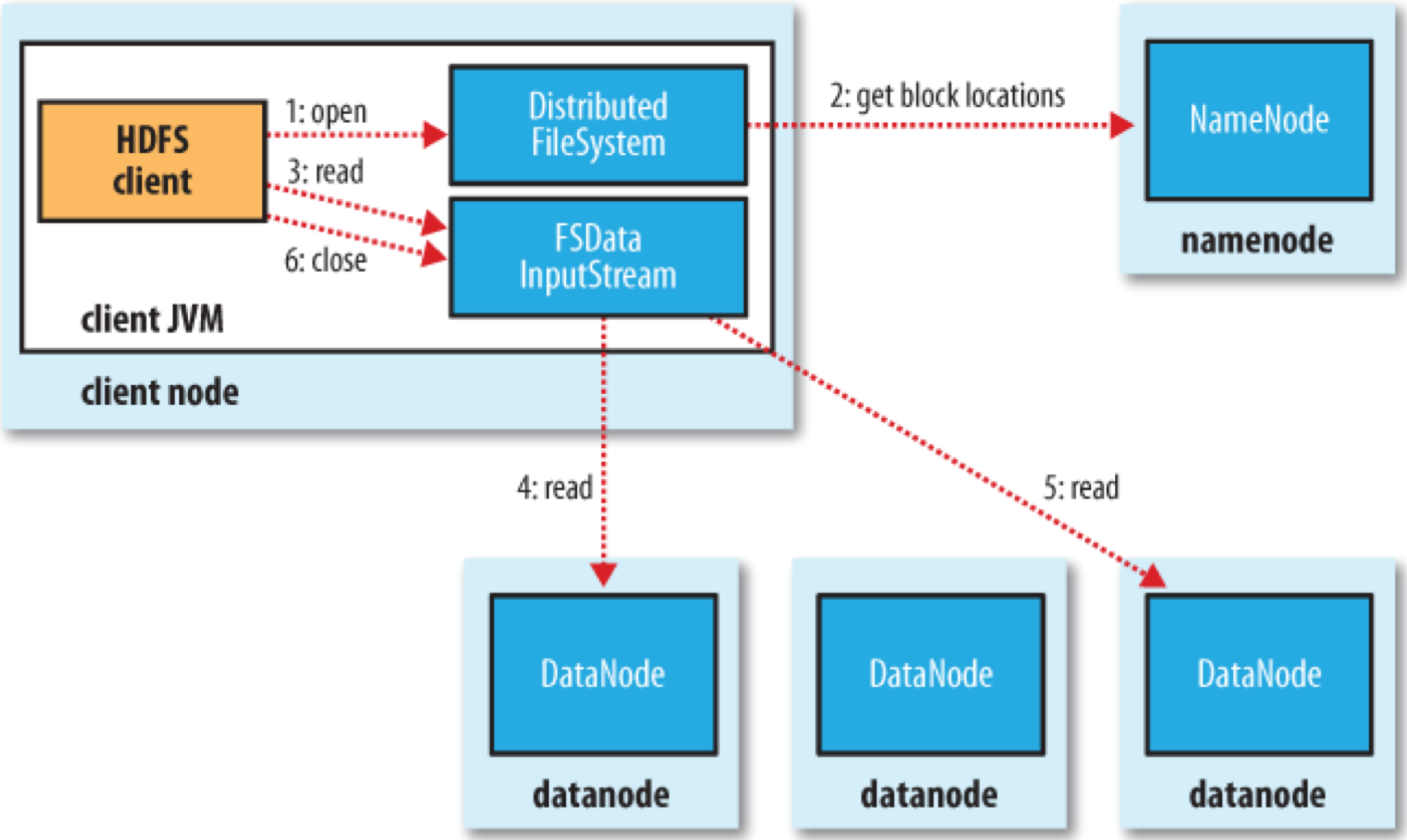
Overview

- This Lecture
  - Distributed FS and Distributed OS
  - Source: Fault tolerance under Unix

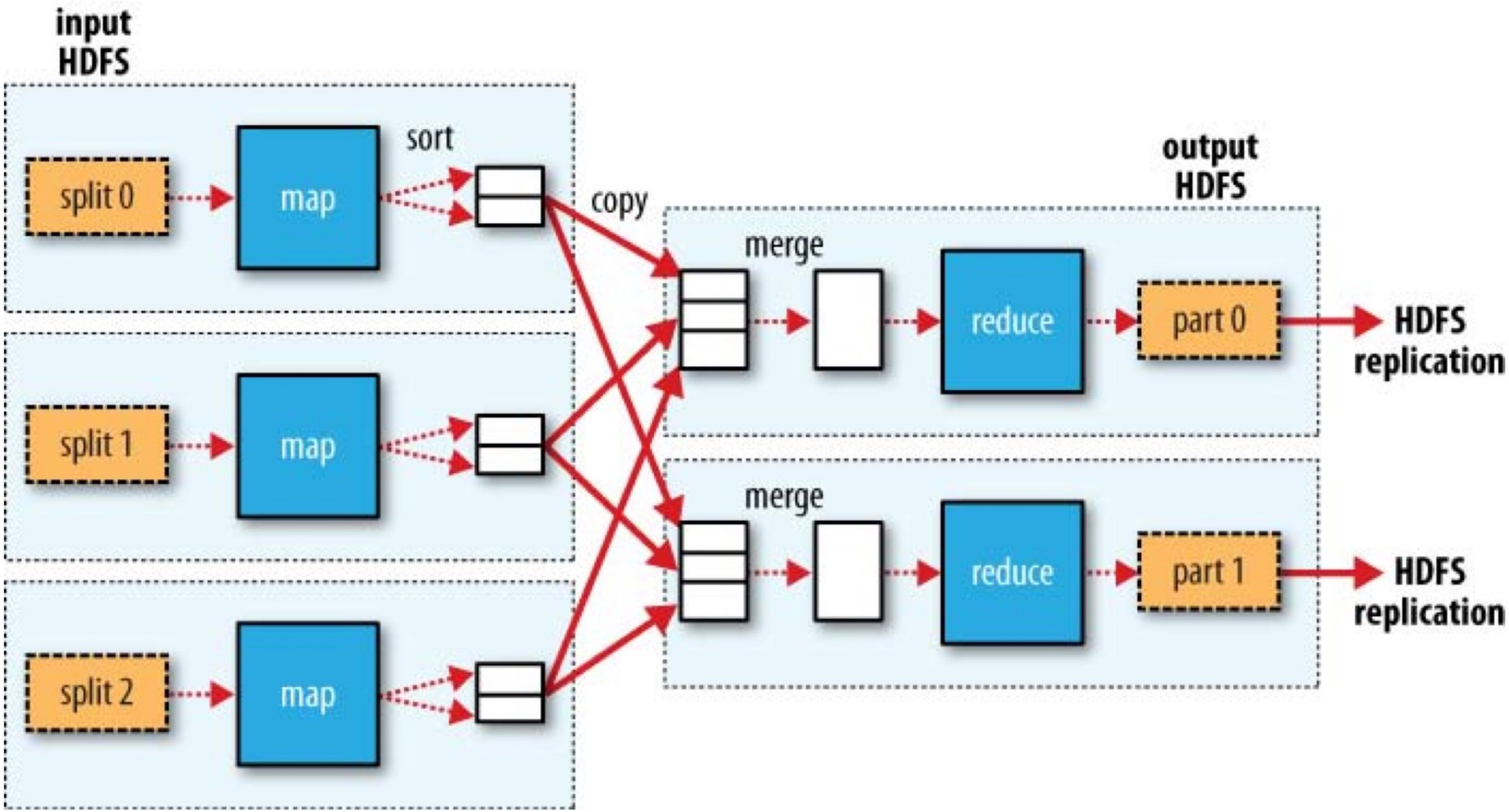
GFS (Google FS)



Hadoop DFS


HDFS client
1: open
3: read
6: close
Distributed FileSystem
FSData InputStream
client JVM
client node
2: get block locations
NameNode
namenode
4: read
5: read
DataNode
datanode
DataNode
datanode
DataNode
datanode

MapReduce


input
HDFS
split 0
split 1
split 2
map
map
map
sort
copy
merge
merge
reduce
reduce
output
HDFS
part 0
part 1
HDFS
replication
HDFS
replication

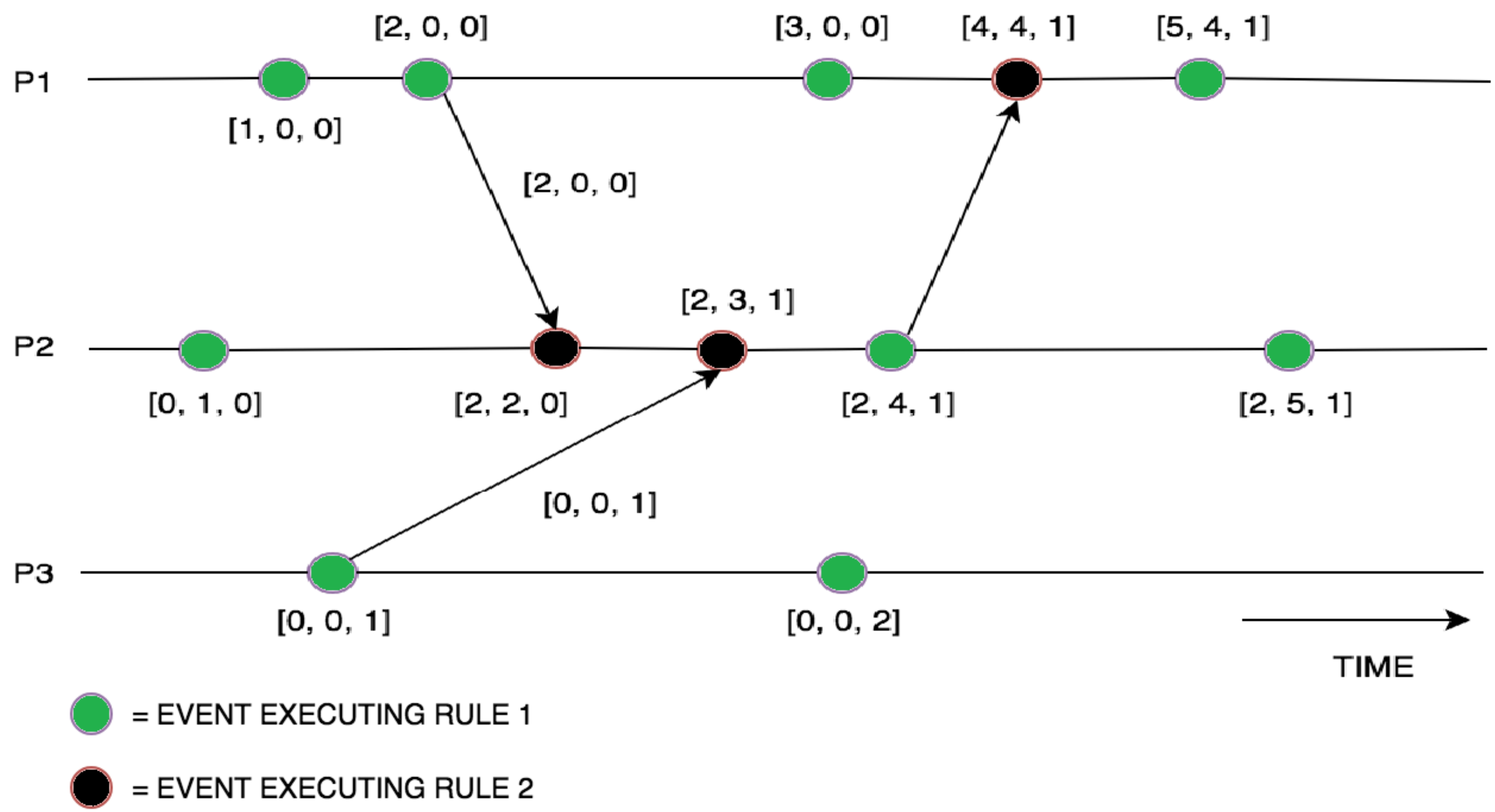
Background

- Fault tolerance with multiple machines in a distributed environment
- Uses IPC and virtual memory to help achieve fault tolerance
- Intended applications
  - Large multi-user applications such as airline reservation and bank transactions
  - Lots of interactive processes using FS
  - Need more power from multiple machines
  - Need more fault tolerance to against HW crashes
    - In 1980s, it was unheard of for computers to stay up for years

Vector time

- Used in distributed systems for causal ordering
- Use a vector to represent the local time of each node in a distributed system.
- Comparing the vector time allows us to know which events are ordered and which events are concurrent.

Vector Time



P1
P2
P3
[2, 0, 0]
[3, 0, 0]
[4, 4, 1]
[5, 4, 1]
[1, 0, 0]
[2, 0, 0]
[2, 3, 1]
[0, 1, 0]
[2, 2, 0]
[2, 4, 1]
[2, 5, 1]
[0, 0, 1]
[0, 0, 1]
[0, 0, 2]
TIME
= EVENT EXECUTING RULE 1
= EVENT EXECUTING RULE 2

General approach

- Up to 16 machines
  - Each with CPU, memory, some with disk
- A broadcast bus connects all machines
  - Maybe very fast and uses dedicated HW
- FS resides on a “root” machine (pair)
- There are multiple server processes
  - Process server where global state such as a list of processes is kept, residing on the “root”
  - File server for all file accesses, residing on “root”
  - Page server manages virtual memory backing store
  - TTY server, raw server

General approach (cont.)

- IPC over bus is used to communicate with servers and processes on other machines
  - E.g. pipe(), fork(), wexec()
  - Fork() require IPC to notify the process server
- Each machine runs a Unix-derived kernel
  - local Unix kernel creates processes, memory, local scheduling, with three processors.
  - Library turns many system calls into IPC
- There is no shared memory among machines
  - The paper's system is an unusual piece of HW
  - Today we have multicore/SMP with shared memory and LAN with message passing

Fault tolerance problem

- The problem faced by the paper
  - Not particularly mentioned
  - Obviously each machine HW is not reliable
  - HW is designed so that each machine fails independently (unlike SMP)
  - SW faults are not the target
- Hard part of the problem
  - Parts of the system can fail (one machine or bus or disk)
  - It is much harder to deal with partial failure than whole failure

The goals of the paper

- Survive any single hardware failure
  - HW has at least two of everything interchangeable
  - Two of every process
- Harness CPUs to increase performance as well as fault tolerance
- Can run ordinary existing Unix applications
- Look like a single large Unix machine
- Fault tolerance and recovery are transparent to applications

Outline of the design

- Have a second copy of each process
  – A “backup” on a different machine
  – Record info about the state of the primary process for backup to use
  – If the primary’s machine crashes, use recorded primary state to make backup equivalent to primary. Then run the backup as the primary
  – When the primary is fixed and restarts, make it backup. Maybe it becomes primary later

Challenges

- How to start backup with state equivalent to the primary's last state?
- How to avoid inconsistency during changeover?
- How to do it all efficiently?
- How to keep it all invisible to applications?

Techniques

- Checkpointing ("sync")
  - Sync primary process' memory periodically
- History recording
  - Record all primary's messages after most recent sync
- Backup process does NOT execute along with the primary
  - Just record the primary's memory and recent messages
  - No high workload on backup's machine
- Why not just "sync"?
  - Memory checkpoint does not record all we need
  - Kernel state and messages sent will make the primary different from the backup

Discussion

- Why not just record all messages?
- The primary sends the pages to the page server when syncing
  - Why not just send to the backup?
- What pages does the primary send when syncing?
- Can the primary execute while syncing?
- What if the primary pages out to the page server between syncs?
- Can we avoid recording messages after sync?
  - E.g. sync all processes and roll them all back on a failure
  - Hard to implement, have to deal with external I/O

Basic idea

- After backup start from the latest memory sync
  - Execute the backup from the sync point
  - Feed it the recorded IPCs the primary received (after sync)
  - Ignore the outgoing IPC
- Why does it work?
  - Assume no source of non-determinism
  - if backup runs the same program as the primary, has same starting state, gets the same IPCs in the same order, it should remain identical
  - It is crucial that all inter-process interaction is via IPC
    - No shared memory between processes
    - Inter-process interaction within a machine should be recorded.

IPC message

- How to record primary's IPC messages?
  - The bus is broadcast, so all machines see all messages
  - Messages have multiple destinations
    - Senders list both primary and backup as destinations
  - Atomicity of reception at the primary and backup is important and guaranteed.
- How to ensure the backup sees the same messages in the same order as the primary?
  - Only one message at a time on the bus
  - The bus is special, very different from LAN
    - LANs do not have total order on messages. Many senders can send at the same time
    - LANs do not have atomic broadcast

Non-determinism

- Why is non-determinism a problem?
  - The primary and the backup may make different decisions
  - They may eventually result in different states, e.g file server may have different contents.
- How to avoid non-determinism?
  - E.g. time() might return differently
  - time() has to be an IPC to the time server
  - Backup sees the reply to primary's IPC request
  - Many system calls are IPCs to some servers
  - Process id is global
  - signal handling is after a primary sync

Playback

- How can the backup know from which message to send real messages?
  – A positive counter is used to record all sent messages since last sync
- What if there is a crash while primary is syncing?
  – Will the page server and the backup disagree about what the last sync is?
  – A sync message is used to confirm to the process' backup, the page server and its backup.
- What about in-kernel state of process?
  – open files, current directory, forked children, etc
  – Open files are expressed as channels to file server
  – Birth notice is used to inform the backup of fork().

Failure detection

- How does the system know if a machine failed?
  - Each machine periodically pings its neighbor in a virtual ring
  - If no response, check if it can talk to anyone else
    - If no, I have failed; if yes, broadcast to all that my neighbor has failed.
- Bus simplifies failure detection
  - No partition
  - Easy to avoid disagreement on if a machine is alive
  - Still it is possible the machine is overloaded and slow to respond. The machine will have a final say anyway.
- When to get backup to replace the primary?
  - After the machine dead message

Recovery

- When a machine dead message gets to the head of the backup's incoming message queue
  - Fetch the latest sync memory snapshot from page server
  - Start executing process
  - Feed it recorded messages since last sync until the machine-dead
  - Ignore its output messages until the counter is 0
  - then let it execute normally
- Note the process has no idea of the above steps
  - Application programmer has no extra work
- Process in-kernel state such as open files is passed in the last sync message
- The roll-forward of messages brings the backup to the latest state of the primary before crash

Discussion

- The old backup is the only copy of the process after recovery
  - Cannot tolerate another failure
  - Must restart the primary as soon as possible
- Is there a problem with externally visible I/O
  - For example, primary updates bank account, then tells external client "done”
  - What if primary crashes just as it is sending the "done"?

Why not the system today?

- HW trends have not favored this approach
  - CPU is much faster than networks/buses, making the close coupling unattractive
  - Much cheaper to use off-the-shelf hardware, e.g. ordinary server and LAN
- Semantics are too strict for many applications
  - For example, email servers do not need to be identical
  - They need to be in different rooms in case of power failure
  - It is ok for one server to receive an email and forward to the other later
  - It is also ok for both receive emails and reconcile later
  - There is high cost for keeping the required semantics

Why not the system today? (cont.)

- Transparency is not that important
  - Programmers don’t care much about it, willing to work to get fault tolerance
  - Being non-transparent results in simplification and efficiency
  - For example, modern database design
    - Use back-end storage server
    - Use many front-end servers to handle many clients
    - Front-end servers only have “soft state”, while real state is in the back-end server and the front-end servers use transactions to read and write DB in order to deal with crashing
    - If a front-end crash during a transaction, the transaction will take no effect, and the client just retry at another front-end
    - The back-end DB server only needs to replicate data, not process execution state.

read()

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
        struct file *file;
        ssize_t ret = -EBADF;
        int fput_needed;

        file = fget_light(fd, &fput_needed);
        if (file) {
                loff_t pos = file_pos_read(file);
                ret = vfs_read(file, buf, count, &pos);
                file_pos_write(file, pos);
                fput_light(file, fput_needed);
        }

        return ret;
}
```

vfs_read()

- ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
- {
- ssize_t ret;
-
- if (!(file->f_mode & FMODE_READ))
- return -EBADF;
- if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
- return -EINVAL;
- if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
- return -EFAULT;
-
- ret = rw_verify_area(READ, file, pos, count);
-

vfs_read()

```
if (ret >= 0) {
        count = ret;
        if (file->f_op->read)
                ret = file->f_op->read(file, buf, count, pos);
        else
                ret = do_sync_read(file, buf, count, pos);
        if (ret > 0) {
                fsnotify_access(file->f_path.dentry);
                add_rchar(current, ret);
        }
        inc_syscr(current);
}

return ret;
}
```