TECHNISCHE
UNIVERSITÄT
DRESDEN

Influential Operating Systems Research

Fault Tolerant Systems

Matthias Hille, designed by Björn Döbel, TU Dresden OS Group

Dresden, 05.06.2018

Outline

The Tandem NonStop System

Replication for Fun and Profit

Bugs in Modern Operating Systems

J. Gray:

Why Do Computers Stop and What Can Be Done About It?,

Tandem Technical Report, 1985

Once upon a time...

- The advent of *online transaction processing*
  - 1964 – IBM SABRE for American Airlines
  - later banking, stock exchange, telephone switches ...
- New requirements
  - Large workloads and data bases (no pun intended)
  - Loss of actual money if the system goes down

Once upon a time...

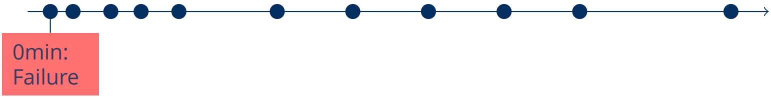
- The advent of *online transaction processing*
  - 1964 – IBM SABRE for American Airlines
  - later banking, stock exchange, telephone switches ...
- New requirements
  - Large workloads and data bases (no pun intended)
  - Loss of actual money if the system goes down

TANDEMCOMPUTERS

- Founded 1974
- NonStop high availability computers
- Acquired by Compaq, later by HP

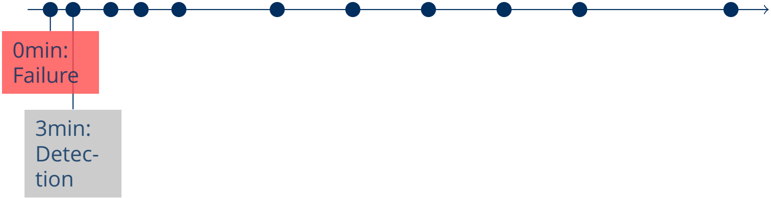
Anatomy of a Failure

“Conventional, well-managed transaction processing systems fail about once every two weeks.”



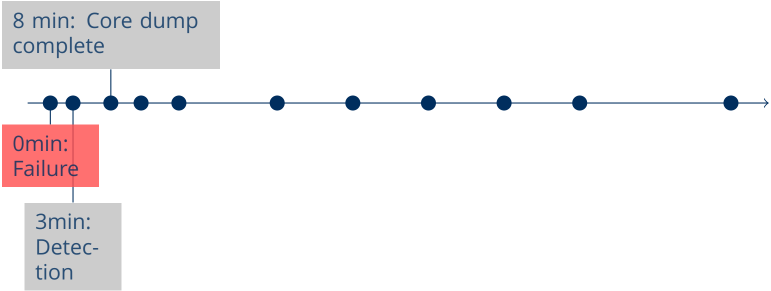
Anatomy of a Failure

“Conventional, well-managed transaction processing systems fail about once every two weeks.”



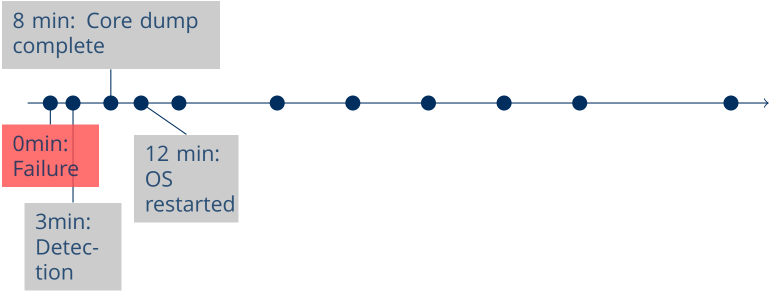
Anatomy of a Failure

"Conventional, well-managed transaction processing systems fail about once every two weeks."



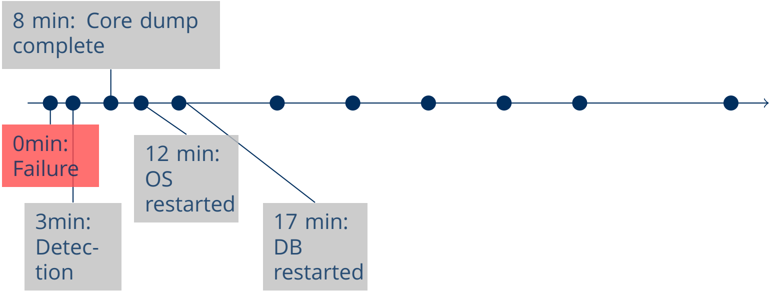
Anatomy of a Failure

"Conventional, well-managed transaction processing systems fail about once every two weeks."



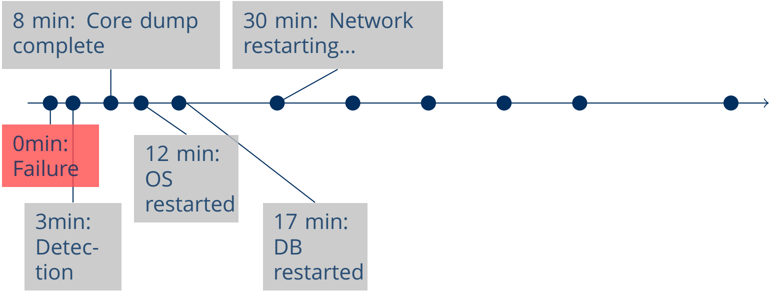
Anatomy of a Failure

"Conventional, well-managed transaction processing systems fail about once every two weeks."



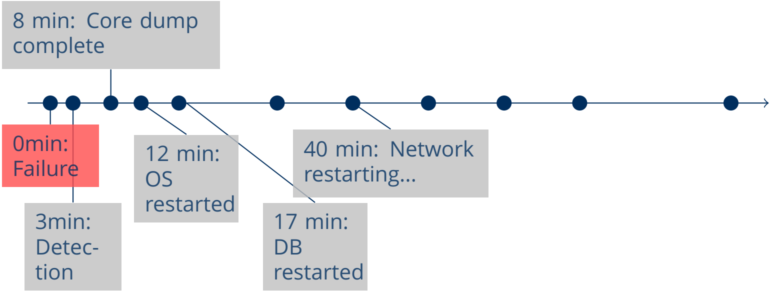
Anatomy of a Failure

“Conventional, well-managed transaction processing systems fail about once every two weeks.”



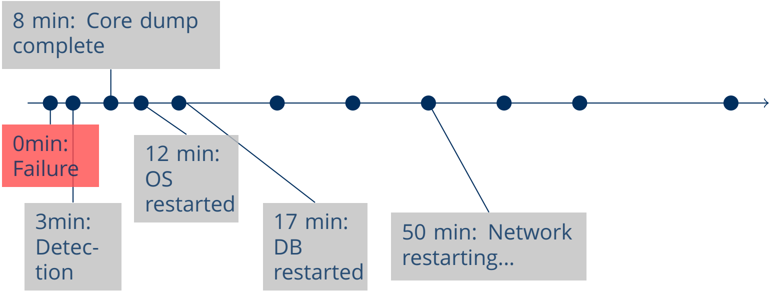
Anatomy of a Failure

"Conventional, well-managed transaction processing systems fail about once every two weeks."



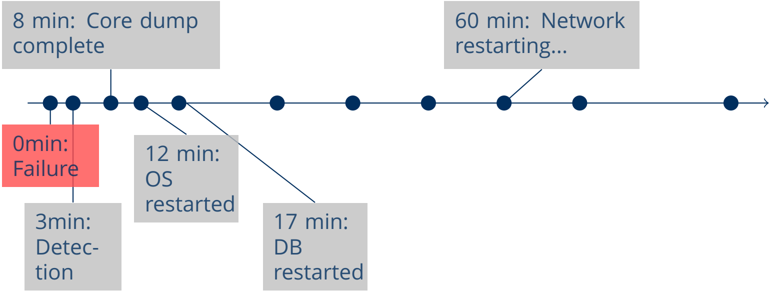
Anatomy of a Failure

"Conventional, well-managed transaction processing systems fail about once every two weeks."



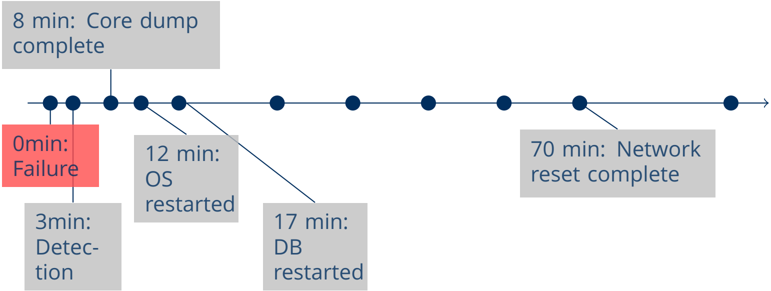
Anatomy of a Failure

“Conventional, well-managed transaction processing systems fail about once every two weeks.”



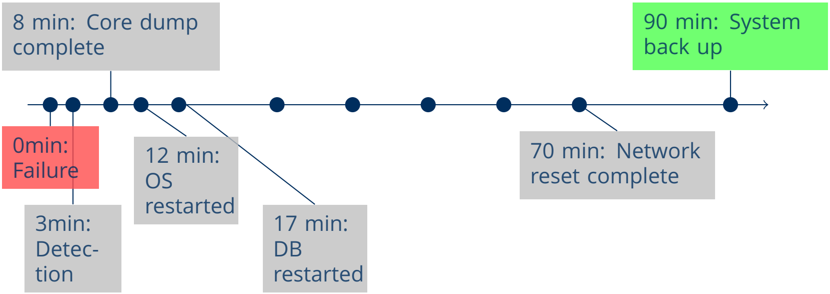
Anatomy of a Failure

"Conventional, well-managed transaction processing systems fail about once every two weeks."



Anatomy of a Failure

“Conventional, well-managed transaction processing systems fail about once every two weeks.”



TECHNISCHE UNIVERSITÄT DRESDEN

Definitions

Fault Model

Defines the expected behavior of faulty components:

- **Fail-stop:** Faulty components do not produce output.
- **Soft failures:** Recovery consists of replacing hardware or restarting software.

Definitions

Fault Model

Defines the expected behavior of faulty components:

- **Fail-stop:** Faulty components do not produce output.
- **Soft failures:** Recovery consists of replacing hardware or restarting software.

Metrics

Mean Time Between Failures: **MTBF**
Mean Time To Repair: **MTTR**

Definitions (2)

Availability

Do the right thing within a specified amount of time.

$$\text{Availability} \quad := \quad \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

Definitions (2)

Availability

Do the right thing within a specified amount of time.

$$\text{Availability} \quad := \quad \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

Reliability

Never do the wrong thing.

Reliability: Design Principles

- Decompose system into hierarchical, isolated **modules**.
  - They actually cite microkernels:
    P. Brinch-Hansen: **The Nucleus of a Multiprogramming System**, CACM 1970

Reliability: Design Principles

- Decompose system into hierarchical, isolated **modules**.
  - They actually cite microkernels:
    P. Brinch-Hansen: **The Nucleus of a Multiprogramming System**, CACM 1970
- Design modules to have an MTBF of more than a year.

Reliability: Design Principles

- Decompose system into hierarchical, isolated **modules**.
  - They actually cite microkernels:
    P. Brinch-Hansen: **The Nucleus of a Multiprogramming System**, CACM 1970
- Design modules to have an MTBF of more than a year.
- Make modules **fail-fast**: Either do the right thing or stop immediately.

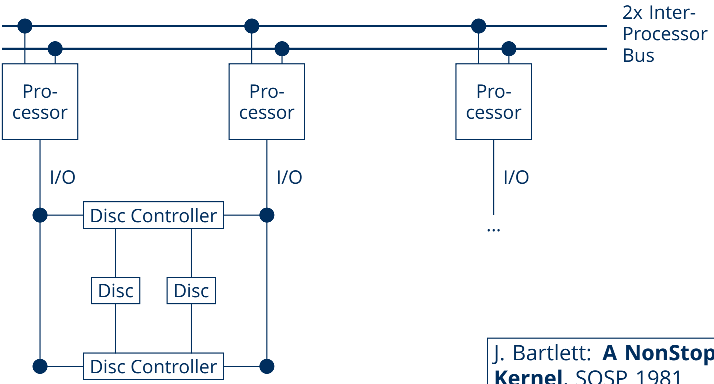
Reliability: Design Principles

- Decompose system into hierarchical, isolated **modules**.
  - They actually cite microkernels:
    P. Brinch-Hansen: **The Nucleus of a Multiprogramming System**, CACM 1970
- Design modules to have an MTBF of more than a year.
- Make modules **fail-fast**: Either do the right thing or stop immediately.
- Detect module failures using **watchdogs** or **heartbeat messages**.

Reliability: Design Principles

- Decompose system into hierarchical, isolated **modules**.
  - They actually cite microkernels:
    P. Brinch-Hansen: **The Nucleus of a Multiprogramming System**, CACM 1970
- Design modules to have an MTBF of more than a year.
- Make modules **fail-fast**: Either do the right thing or stop immediately.
- Detect module failures using **watchdogs** or **heartbeat messages**.
- **Redundancy:** Configure extra modules that can take over in case of failure.

NonStop: Hardware



J. Bartlett: **A NonStop Kernel**, SOSP 1981

NonStop: Kernel Services

Per node: memory+process manager

Fault-tolerant messaging: RPC-style programming model

- Abort calls at any time

Packet protection

- Sequence numbers
- Data Checksums
- Timeouts: resend over alternative channel
- Batched acknowledgments: dual function as heartbeat

NonStop: Software

Software services implemented as **process pairs**

- Primary: handles all requests
- Backup: steps in if primary failure is detected
  a) Initiate restart of primary
  b) Launch new backup process
- Primary + Backup run on different processors.
- OS maintains Primary/Backup table.

NonStop: Software

Software services implemented as **process pairs**

- Primary: handles all requests
- Backup: steps in if primary failure is detected
    a) Initiate restart of primary
    b) Launch new backup process
- Primary + Backup run on different processors.
- OS maintains Primary/Backup table.

How do we keep the backup up-to-date?

NonStop: Syncing Primary+Backup

1. Lock-stepping
   – Process all requests at both partners step-by-step.
   – Will catch hardware errors, but no software ones.
2. State Checkpointing
   – Primary sends all requests and replies to backup.
   – Requires additional programming effort.
3. Delta Checkpointing
   – Instead of sending every physical request, send diffs of service state to the backup.

NonStop: Syncing Primary+Backup (2)

4. Automatic Checkpointing
   - Log all messages, only replay in case of failover.
   - If state grows to large, send physical state update.

5. Persistent Processes
   - Do not send updates at all!
   - Instead, backup wakes up in NULL state.
   - But service state needs to always be consistent!
     a) Every successful request leaves the service state consistent.
     b) Every failing request does not modify service state at all.

NonStop: Syncing Primary+Backup (2)

4. Automatic Checkpointing
   – Log all messages, only replay in case of failover.
   – If state grows to large, send physical state update.

5. Persistent Processes
   – Do not send updates at all!
   – Instead, backup wakes up in NULL state.
   – But service state needs to always be consistent!
     a) Every successful request leaves the service state consistent.
     b) Every failing request does not modify service state at all.

But isn't that...?

Transactions!

- **A**tomicity: all or nothing state modification (commit or abort)
- **C**onsistency: always work on consistent state (even during concurrent transactions)
- **I**ntegrity: all state transformations need to be correct
- **D**urability: commited transactions remain persistent

Why is this good for reliability?

- No state inconsistencies
- Builtin abort + undo upon failure
- No state checkpointing between primary and backup

F. Schneider:

Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial,

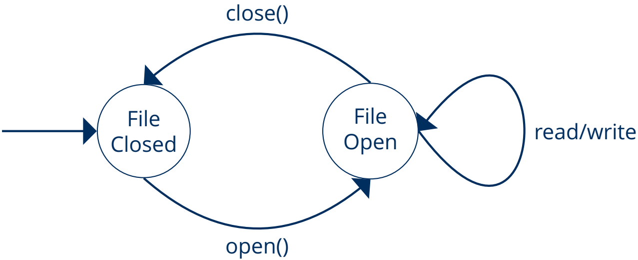
ACM Computing Surveys, 1990

More Fault Models

- **Byzantine Failure:** Faulty components produce arbitrary, potentially malicious output.
- **Common Cause Failures:** Multiple components fail at the same time because they are subject to the same cause.

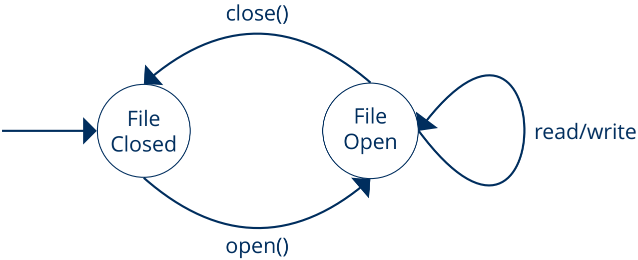
Software Model

An application implements a service in the form of a **state machine**.



Software Model

An application implements a service in the form of a **state machine**.



Can every application be implemented as a state machine?

State Machine Properties

- **Sequentiality:** Requests are processed atomically.
- **Determinism:** The same sequence of requests produces the same output.
- **Independence from time:** The timing of requests does not influence state transitions.

Tolerating Independent Failures

T Fault Tolerance

A system is **t fault tolerant** if it satisfies its specification provided that no more than t of its components become faulty during some interval of interest.

Tolerating Independent Failures

T Fault Tolerance

A system is **t fault tolerant** if it satisfies its specification provided that no more than t of its components become faulty during some interval of interest.

Replication

T Fault Tolerance can be achieved by running multiple independent replicas of a state machine.

- Fail-stop: $T + 1$ replicas are needed.
- Byzantine: $2T + 1$ replicas and majority voting
- Common cause: Physically/geographically distribute replicas.

Implementing State Machine Replication

- Replicas need to be coordinated:
  - **Agreement:** All replicas need to see all requests.
  - **Order:** All replicas process requests in the same order.
- Relaxations may improve performance:
  - Read-only requests in fail-stop systems need only be serviced by a single replica.
  - Commutative requets may be processed in any order.
- Coordination problems:
  - Requests may get lost.
  - Requests may overtake each other.

Implementing Ordering

It’s simple:

- Assign requests unique identifiers.
- Ensure total ordering of UIDs is possible.
- Process requests in order of their IDs.

Not quite...

- How to assign IDs?
- When does a replica know that a request reached all other replicas?

Stability

Stability

A request is defined to be **stable** at state machine $\mathrm{SM_i}$ once no request from a correct client and bearing a lower unique identifier can be subsequently delivered to $\mathrm{SM_i}$.

Order Implementation

A replica next processes the stable request with the smallest unique identifier.

Ordering with Logical Clocks

Assign each event e a timestamp $T(e)$, so that if we have two events e and f and e might be responsible for causing f, then $T(e) < T(f)$.

L. Lamport: **Time, Clocks and the Ordering of Events in a Distributed System**, CACM, 1978

TECHNISCHE UNIVERSITÄT DRESDEN

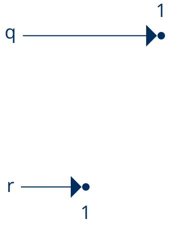
Ordering with Logical Clocks

Assign each event e a timestamp $T(e)$, so that if we have two events e and f and e might be responsible for causing f, then $T(e) < T(f)$.

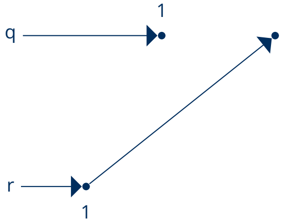
L. Lamport: **Time, Clocks and the Ordering of Events in a Distributed System**, CACM, 1978

- Each process p is assigned a counter T_p.
- Each message m is augmented with the value of T_p when m was sent by p.
- T_p is then updated as follows:
  1. Each event at p increments T_p.
  2. When receiving a message, the receiver r updates $T_r := \max(T_m, T_r) + 1$.

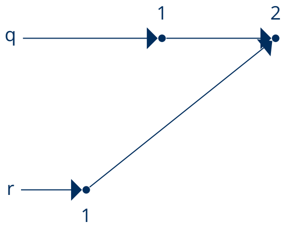
Logical Clocks: Example



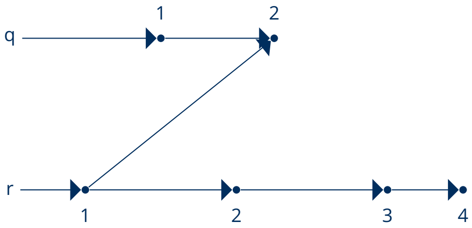
Logical Clocks: Example



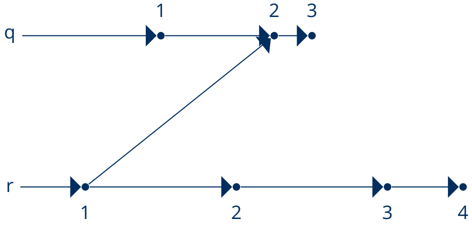
Logical Clocks: Example



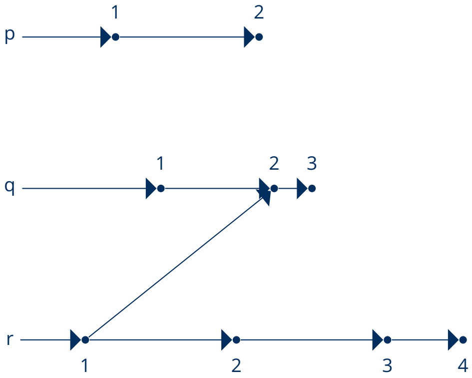
Logical Clocks: Example



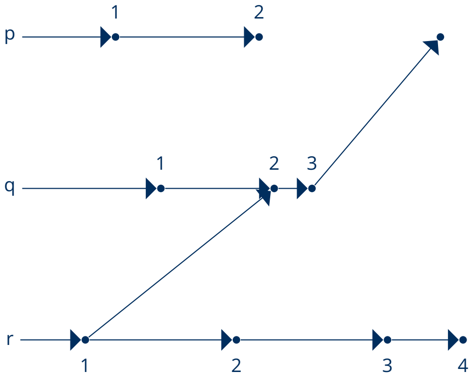
Logical Clocks: Example



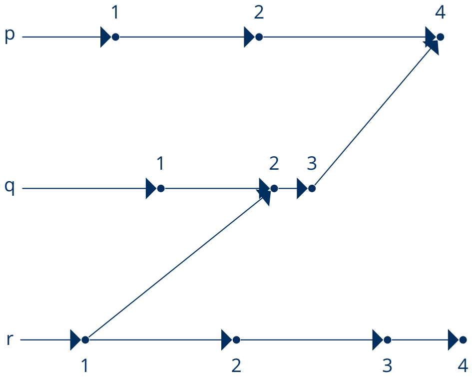
Logical Clocks: Example



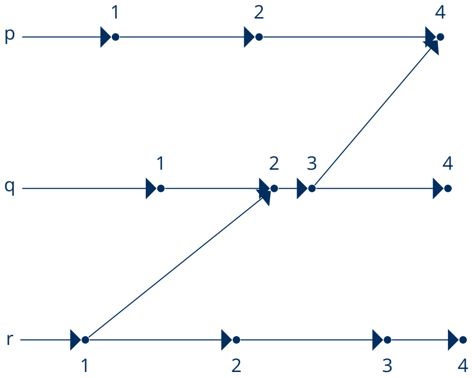
Logical Clocks: Example



Logical Clocks: Example



Logical Clocks: Example



Logical Clocks and Replicas

- **FIFO Channels:** Logical clocks establish send order between any pair of processors.

Logical Clocks and Replicas

- **FIFO Channels:** Logical clocks establish send order between any pair of processors.
- Replica ordering:
  - All processors periodically send heartbeat messages (broadcast!).
  - A request is stable at replica $\mathrm{SM_i}$ if a request/heartbeat with a larger timestamp has been received by $\mathrm{SM_i}$ from every non-faulty processor.

Things to Consider

- Can also integrate stability generation into real-time clock synchronization.
- If sync traffic is a concern, algorithms to generate UIDs with less messages exist.
- The $2T + 1$ rule for byzantine faults only works for the case of a correct voter!
  - So we might want to replicate voters
    see Berninck: **NonStop: Advanced Architecture**, DSN 2005
  - Otherwise this becomes the **Byzantine Generals Problem**, which is only solvable with $3T + 1$ participants
    see Lamport, Pease, Shostak: **The Byzantine Generals Problem**, 1982

N. Palix et al.:
Faults In Linux: Ten Years Later,
ASPLOS 2011

Lecture on Experiments

- Document system and configuration
- Publish and keep raw data, setups, ...
- Experiments must be repeatable by others.

Repeating Experiments in the Real World

The Original:
A. Chou et al. **An Empirical Study of Operating System Errors**, SOSP 2001

- Static code analysis of Linux 1.0 – 2.4.
- Device drivers 3x more likely to contain bugs than rest of kernel code.

Hypothesis:
10 years of research on improving device driver quality should have had an impact.

Validation:
Repeat Chou's experiments with Linux 2.6 kernels.

Static Source Code Analysis

Check potentially NULL pointers returned from routines.

```
my_data_struct *foo =
      kmalloc(10 * sizeof(*foo), GFP_KERNEL);
foo->some_element = 23;
```

Do not use freed memory

```
free(foo);
foo->some_element = 23;
```

Var

Do not allocate large stack variables (>1K) on the fixed-size kernel stack.

```
void some_function()
{
    char array[1 << 12];
    char array2[MY_MACRO(x,y)]; // not found
    ...
}
```

Inull

Do not make inconsistent assumptions about whether a pointer is NULL.

```
void foo(char *bar)
{
    if (!bar) { // IsNull
        printk("Error: %s\n", *bar);
    } else {
        printk("Success: %s\n", *bar);
        if (!bar) { // NullRef
            panic();
        }
    }
}
```

LockIntr

Release acquired locks; do not double-acquire locks. Restore disabled interrupts.

```
void foo() {
   DEFINE_SPINLOCK(l1); DEFINE_SPINLOCK(l2);
   unsigned long flags1, flags2;

   spin_lock_irqsave(&l1, flags1);
   spin_lock_irqsave(&l2, flags2);
   // double acquire:
   spin_lock_irqsave(&l1, flags1);
   ..
   spin_unlock_irqrestore(&l2, flags2);
   // unrestored interrupts for l1/flags1
   // + unreleased lock l1
}
```

Range

Always check bounds of array indices and loop bounds derived from user data.

```
int index = -1;
int n = copy_from_user(&index, userptr,
                       sizeof(index));
if (!n) {
  kernel_data[index] = 0x0815;
}
```

Size

Allocate enough memory to hold the type for which you are allocating.

```
typedef int       myData;
typedef long long yourData;

yourData *ptr = kmalloc(sizeof(myData));
```

Lines of Code

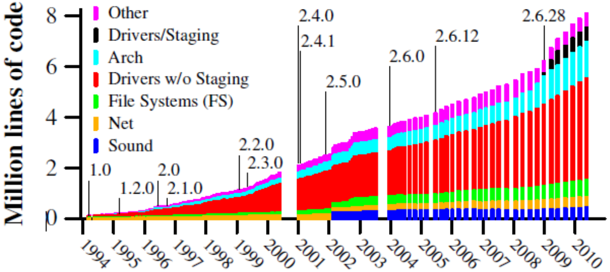
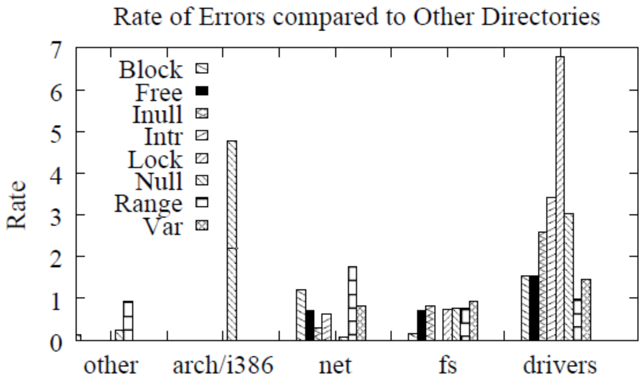
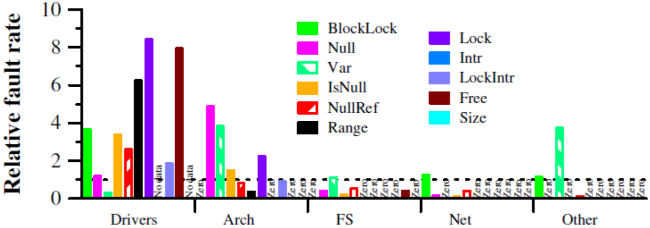


Figure 1. Linux directory sizes (in MLOC)

Fault rate per subdirectory



Fault rate per subdirectory



(b) Rate of faults compared to all other directories

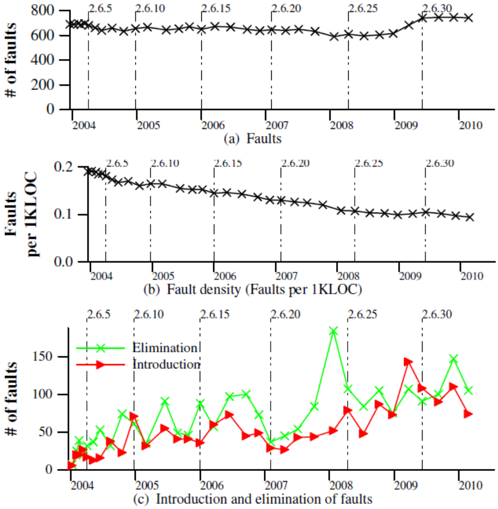


Figure 6. Faults in Linux 2.6.0 to 2.6.33

Crying for help

...Because Chou et al.'s fault finding tool and checkers were not released, and their results were released on a local web site but are no longer available, it is impossible to exactly reproduce their results on recent versions of the Linux kernel...

In laboratory sciences there is a notion of experimental protocol, giving all of the information required to reproduce an experiment...

Crying for help

...Chou et al. focus only on x86 code, finding that 70% of the Linux 2.4.1 code is devoted to drivers. Nevertheless, we do not know which drivers, file systems, etc. were included...

...Results from Chou et al.'s checkers were available at a web site interface to a database, but Chou has informed us that this database is no longer available. Thus, it is not possible to determine the precise reasons for the observed differences...

Summary

- Custom-tailoring for fault tolerance: it's getting harder as systems grow more complex.
- Distributed systems fault tolerance: it's running the cloud (tm).
- Device drivers are still an issue.