Consider a set class implemented as a linked list.

One can create objects of class set (myset).

Then one can access that class using member functions through a driver class called SetDriver.
SetDriver

// SetDriver.java
import java.io.*;
import java.util.Vector;
public class SetDriver {
    public static void main(String argv[]) {
        ...
        Set myset = new Set();

        myset.add("foo");
        if (myset.member("bar")) {
            myset.delete("bar");
        } else { myset.delete("barbar"); }
        ...
    }
}
©Dennis Shasha
Set Class

Set(Set other) throws SetException {...
boolean member(String el) {
    // search linked list
void add(String el) {
    if (false == member(el)) {
        nodes = new SetEl(el, nodes);
    }
void delete(String el) {...
Set Element Class

class SetEl {
    SetEl(String el, SetEl nextNode) {
        string = el;
        next = nextNode;
    }

    String string;
    SetEl next;
}
Concurrent Activity

- Invoke each thread by creating a new thread object, passing values to the initialization function, then invoking run.

- How to set up threads?
  Set up a class that extends thread, providing an initialization function (setThread) and a run() function.
Using the Thread Class

```java
public class SetDriver {
    public static void main(String argv[]) {
        ...
        Set myset = new Set();
        setThread s = new setThread(myset,"foo");
        s.start();
        setThread s2 = new setThread(myset,"foo2");
        s2.start();
        ...
    }
}
```
Thread Class

class setThread extends Thread {
    Set a;
    String s;
    setThread(Set in, String instring) {
        a = in;
        s = instring;
    }

    public void run() {
        a.add(s);
        a.delete(s);
        a.printall();
    }
}

©Dennis Shasha
Need for Synchronization

- If there were no concurrency (i.e. if one thread executed at a time), then each printall would print out nothing in our example.

- With concurrency though in fact, we get intermediate results in which the list has many elements.

- To achieve thread serializability, we establish a synchronized lock.
Using the Lock

public void run() {
    while(a.getlock() == 1) {
        try {Thread.sleep(10);} 
        catch (InterruptedException e) {} 
    }
    a.add(s); 
    try {Thread.sleep(1000);} 
    catch (InterruptedException e) {} 
    a.delete(s); 
    a.printall(); 
    a.resetlock(); 
}
Implementing the Lock

// if unlocked, lock it and return 0.
// if locked, then return 1.
synchronized int getlock() {
    if(lock == 0) {
        lock++;
        return 0;
    } else {return 1;}
}

// unlock this.
synchronized void resetlock() {
    lock = 0;
}
Summary of Java Concurrency Facilities

- Threads give concurrency without control. (Single Java statements may be interrupted by other threads.)

- Synchronized class members give serializability for a single function invocation.

- From that, one can build synchronization over many function invocations.
Impossibility Results

*Making any progress proving things impossible?* William Greatbatch (inventor of the pacemaker) to Steve Cook (discoverer of NP-completeness).

- “Impossibility of Distributed Concensus with One Faulty Process” M. J. Fischer, N. A. Lynch, and M. S. Patterson JACM April 1985

- Clean failures and slow messages.
Model

- Messages take unbounded delays though they eventually arrive.

- Local state: result of initial value plus messages received.

- Pretend that configurations advance one message receipt at a time. Simultaneity can’t be detected anyway.
Bivalence

• To achieve consensus, a protocol must achieve the following for a set of processes. Given that each process receives a 1 or 0 input, all non-failed processes must agree (all must conclude 1 or all must conclude 0) and they all must agree on a value at least one of them receives as an input.

• Configuration is bivalent, if either 1 or 0 can be the final outcome. Configuration is univalent at 0 if it will result in 0 even with one failure (analogous for univalent at 1).
Bald Man’s Lemma

- There exists a bivalent initial configuration.

- Idea: All 1s should be univalent at 1.
  All 0s should be univalent at 0.
  Starting from all 1s, start changing inputs to 0, at some point your protocol should switch to univalent at 0. What if there is a failure just at that border? You don’t know what the failed process received. That configuration must be bivalent.
One Process Lemma

- Lemma: If we start in a bivalent initial configuration and eventually arrive at a univalent configuration, then the valence of a configuration depends on a single process.

- By contradiction: let $B$ be the last bivalent configuration. Let $U_0$ be a zero-valent configuration resulting from $B$ and a message $m_0$. Let $U_1$ be a one-valent configuration resulting from $B$ and a message $m_1$. 
How Do these Configurations Differ?

- If \( m_0 \) and \( m_1 \) go to different processes, then the order of their arrival doesn’t matter. (Local state assumption.)

- So, \( m_0 \) and \( m_1 \) must go to the same process \( p \) and order of the messages matters.
Now, What if p Fails?

- If p fails, then the configurations U0 and U1 are indistinguishable, so there cannot be an unambiguous univalent outcome from these two messages.

- Bottom line: can’t guarantee both liveness and safety of consensus.
Broadcast and Consensus

• Suppose you had an algorithm that could guarantee safety and liveness of ordered broadcast (all sites accept messages in same order).

• Consensus algorithm: everyone broadcasts message. First one received is desired value.

• So, ordered broadcast can be safe or live but not both. In practice, safety is what one wants.
First Principles of Fault Tolerance

- Don’t do anything bad on big failures (safety).
- Mask small failures (availability).
- Example: traffic light flashes red when controller fails.
Classification of Failures

- response failure = bad output (plus(2,2) returns 5)

- omission failure = no response, e.g., message loss.

- crash failure = no response ever again. (Hard to distinguish from omission.)

- timing (performance) failure = too early or too late.
Building Blocks

- service = desired behavior

- service spec = function from invocation X history to output

- server = agent that performs a service
Masking and Dependence

- masking $=$ service doesn’t fail even though a server might.
  Example: triplicate LAN cable. Each cable is a server. Communication is the service.

- depends on $=$ server s depends on service t if correct behavior of s requires that t meets its service specification.
  When s depends on t, it is often the case that s calls t, but there are exceptions: e.g. air traffic control depends on radar sensor.
Example: Transmission System

- Suppose system may suffer omission failures, but no performance failures. So, all request/replies that arrive complete within time T. In that case, if client doesn’t get reply within T, then sender can send the same message again without worrying about receiving replies to old messages (sequence numbers are unnecessary).

- In case where performance and omission failures are both possible, need sequence numbers or some other mechanism to ensure that replies are tied to specific requests.
Masking Techniques

- Hierarchical masking: user of service compensates for failure of lower service, e.g. by retrying.

- Group masking: group of servers hide failure of single server. If servers are stateless, then response of a service may be the response of the fastest server. That is highly fault tolerant and servers can enter and exit at will.

- If servers have state, must worry about consistency (miss a single message and you may do the wrong thing).
Example: main memory subsystem

You want to tolerate a single response failure (single bit flip), then have a choice:

- Buy two memories with parity checking, write to both, read whichever doesn’t suffer from a parity error.

- Buy three memories without parity checking, write to all three, use voting on response. May need more.
Example: Reliable Memory System

- Memories have error correcting code for single bit failures. Moreover, each task is divided into a sequence of code segments and the state at the end of each segment is held in two memory modules (MM1 and MM2)

- To update a code segment:
  1. copy one of the memory modules to a third one (MM3)
  2. execute code on MM3 (if fail, stop)
  3. update MM2 and increase state number
  4. update MM1
Preliminary Conclusions

• To eliminate response failures from hardware, duplicate the hardware and use comparison. This gives crash failure semantics (assuming failures are independent).

• There is too much correlation among software errors for voting to work well.

• **Communication is easier to make fault tolerant than persistent state.** Persistent state requires agreement.
Preliminary Lessons for Software

- Software should use simplicity of design, hierarchical decomposition, information hiding, detection and handling of all exceptions, design and code inspections.

- If s depends on t, then exceptions raised in t should be handled in s.
Replicated State Machines

- Database management systems are normally concurrent, but many main memory systems are sequential. Suppose one wants fault tolerance for main memory systems.

- Replicated state machine: each site receives operations in the same order and processes them sequentially. If any site breaks, system just uses other sites.

- No need for a commit protocol, provided there are sequence numbers. Client can continue when at least two sites have received this and all previous operations and client has any return value information. Log is just the set of operations (no after-images).
Problem with Replicated State Machines

- Could be slow if many operations must go to disk.

- ISIS alternative: allow transactions to run concurrently thus allowing multiple concurrent reads from disk. At commit time, make sure they commit in the same order at all replicas. ISIS atomic broadcast ensures that commits will tend to occur in the same order everywhere. Must do value logging.
Operation vs. Value Logging

- Operation logging — log the action. Usually less data than amount of data changed.

- Value logging — log the value image. Gives quicker recovery time.

- Ideal — Operation logging on disk and value logging in memory.
Fault Tolerance for Embarassing Parallelism

- Suppose that you have a set of computations you want to perform that are mutually independent and idempotent (multiple executions of a given task all produce the same result). Compute engines may fail.

- Whitney technique: establish a queue of tasks. Send a task t to a compute engine and then put t at end of queue. When t completes, remove it from the queue.
Idempotency is Critical

- There is no need to distinguish between slow engines and failed ones.

- What if the task implemented “plus 10”? 
Kedem-Palem Idempotency Idea

- Suppose we want to support fault tolerance for arbitrary sequences of operations.

- Sequence begins and writes all updates to a set of memory locations on the “side”.

- Under assumption of CREW model, Calypso writes diffs to side memory locations and writes back to primary.
Forcing Idempotency

- Suppose you are in a distributed environment in which a workflow controller must communicate with several databases, updating all or at least some of them.

- Updates are not normally idempotent.

- What if workflow controller or one of the other databases fails?
Other Database Fails

- Have to hope it will come up whole.

- If not, you can recover it if controller is the only interface the database has and controller keeps a log of its operations.
Workflow Controller Fails

- It must recreate its local state (either from a local operation log or a value log).

- Local state will not include an exact picture of states of databases (e.g., acks may have been lost).

- Don’t want to repeat possibly non-idempotent updates on these databases.
Encode Continuations in Transactions

• When updating the database, record the updating operation in a separate table as part of the transaction.

• Upon recovery, don’t redo an update transaction if already in that operation table.

• Obviously the identifier for the operation must be the same when recovering as the first time around. Application-specific but easy to do with sequential machines.
Ethernet Protocol

- Ethernet is a collision detect protocol for broadcast media.

- If a site wants to send a message, it waits until there is silence. Then it transmits. If another site transmits at the same time, then the first site will detect a collision.

- Each site participating in the collision waits a time chosen uniformly in the range 0 to $T$ (a parameter of the protocol) and then tries again. In exponential backoff, the $T$ chosen can double. Livelock is possible.
Real-time Ethernet

- Developed by Gerard LeLann for the French navy, this protocol was a military secret at one point and is now patented.

- Sites listen all the time. They won’t try to transmit while there is a duel going on unless they are in the dueling group. Initially, there is no duel going on. Suppose a set of sites collide when no duel is going on. These sites form the dueling group.
Bet Your Age

• Assume now that each site has a unique identifier in binary (e.g. an IP address). All those in the dueling group with a 0 in the high order bit (of host address) attempt to send in the next time unit. If there is still a collision, then all those with a 0 in the next high order bit attempt to send in the following time unit.

• The procedure terminates with a winner in log(total number of sites) time. After that, the duel continues among the remaining sites that last collided.
Example

Example: Suppose the following sites collide at time 0: 1, 2, 3, 4, 5, 8, 14. Assume each site has a four bit descriptor.

- Time 1: All sites with a 0 in the high order bit, i.e., 0???, will attempt to transmit. In this example, that means sites 1, 2, 3, 4, and 5.

- Time 2: All sites with 0 in first two high order bits, i.e., 00??, (1, 2, 3) will attempt to transmit.
Example Continued

• Time 3: Sites 000? (site 1) will attempt and succeed since it is alone.

• Time 4: Sites 001? (sites 2 and 3) will attempt (because they know there is no collision having to do with 000?, so each thinks it might be next).

• Result will be a left-to-right traversal of the tree.
Costs and Benefits

- Guaranteed liveness.

- Some loss of efficiency. (Collision at time unit 4 may be optimized away.)

- A cheater would mess things up royally.
Distributed “Real-Time”

- When you hear the words “best-effort,” then you had better not hear “guaranteed.”

  http://www.fokus.gmd.de/step/hgs
Problem

- Set of messages. Each message has a deadline and a number of hops to get through to its destination.

- A hop processes a given message in minimum time \( t_{\text{min}} \) and maximum time \( t_{\text{max}} \).

- Messages queue up: each node can delete messages or reorder its queue.
Goal

• Get packets to destination by deadline. Kill those that won’t make it for sure. What about the others?

• Kill those that are unlikely to make it. Give low priority for those that have large laxities. Give high priority to those who are a little behind.

• Bifurcated priorities: nice idea.
Research Question

• Can anything be guaranteed?


• What if knowledge is purely local?
D-Over: an optimal scheduling algorithm for overloaded real-time systems

Gilad Koren

Dennis Shasha

INRIA, Rocquencourt

and

Courant Institute of Mathematical Sciences,
New York University

◊◊◊

First presented January 17, 1992 at INRIA, Rocquencourt.
Question

Suppose you have a set of tasks you must accomplish with tight deadlines. You realize you cannot accomplish them all.

Question: How should you schedule yourself?

Answer: Algorithm follows (under certain assumptions).
Parameters in a Real-Time System

Task characterized as follows:

- **release time** — point when request for task execution arrives.

- **computation time** — time needed to complete task.

- **deadline** — point before which task should complete to obtain full value.

- **value** — value of task if completed by its deadline.

\[
\text{value density} = \frac{\text{value}}{\text{computation time}}
\]
Model

- **on-line scheduling** — Nothing is known about a task until it arrives.

- **preemption is cheap** — Task switching takes no time.

- **not hard real-time** — Missing a deadline is not catastrophic.

Goal: Achieve best competitive algorithm possible with respect to a clairvoyant scheduler (knows future and has unbounded computation resources).
Previous Results

Definition:
underloaded system — some (possibly clairvoyant) algorithm can schedule all tasks to completion. Otherwise, the system is said to be over-loaded

- D: earliest deadline first.
  schedule the unfinished task with the earliest deadline.

  D performs as well as the clairvoyant scheduler for an underloaded system (i.e. schedules all tasks).
Naively applying D to an overloaded situation can be very bad.

Example:

- All tasks $i$ ($i \geq 0$) have computation time $1 + \epsilon$. One additional task $T$ has computation time $\epsilon$ and deadline 1.

- Task $i$ has release time $i$ and deadline $i + 1 + \epsilon$.

- No value for a task that completes after its deadline.

- Task 0 and task $T$ are released at time 0.

Result: Only $T$ will complete.
First Heuristic

*Useful Work Heuristic:* Don’t schedule a task that cannot possibly finish by its deadline. Otherwise, try to follow earliest deadline first.
First Heuristic Not Enough

Example:

- All tasks have computation time 1.

- Task $i$ has deadline $100 - i/1000$.

- No value for a task that completes after its deadline.

- Task 0 is released at time 0.

  Task $i$ ($99 \geq i \geq 1$) is released at time $i - i/1000$.

Only the task last released will complete. Every other one will be preempted $1/1000$ time units before completing.
Second Heuristic

*Global Useful Work Heuristic:* Schedule a task with earlier deadline if it can finish and it will not prevent some other task from finishing.

If it will prevent some other task from finishing, think carefully.
Figure 1

http://cs.nyu.edu/courses/fall05/G22.2631-001/docs/
Value Densities and Overload

Let $k$ be highest value density. (Minimum value density is 1.)

Example:

- A long task with value density 1 and zero slack time is currently executing.

- Short task appears with value density $k$ and zero slack time.

Only one of these tasks can complete. Do you switch?

---

Definition: Slack time =
time to deadline - remaining computation time
Figure 2

http://cs.nyu.edu/courses/fall05/G22.2631-001/doc
Inherent Limits

In the previous example,

- If you do switch, adversary gives you no more tasks.
- If you don’t, adversary keeps giving you short tasks.

So, cannot possibly be better than $1/k$ as good as clairvoyant.

In fact, can not be better then

$$\frac{1}{(1 + \sqrt{k})^2}$$

Baruah, Koren, Mao, Mishra, Raghunathan, Rosier, Shasha, Wang, 1991
Our Results

- D-over matches the bound of $1/(1+\sqrt{k})^2$. when computation time is known precisely upon task release. Also guarantees 100% of the value during underloaded periods.

- A little lower when computation time is known only within a range.

- Competitive algorithm even in cases where value stays positive after deadline is passed.
Algorithm Works in two modes

- **Underloaded mode** — all currently preempted tasks have time to complete. They are executed in deadline order. Looks like D for those tasks.

- **Overloaded mode** — a task that reaches its latest start time must “duel” with the currently executing task (as well as preempted tasks) to determine whether it should execute or be abandoned.

Enter overloaded mode when a task reaches its LST and is scheduled for executing. Exit overloaded mode when such a task completes.
Underloaded Mode

when a new task $T$ is released:

- if deadline of $T$ precedes deadline of currently executing task
  and
  there is time for $T$ to execute as well as all preempted tasks

  then $T$ executes

else $T$ waits
Overloaded Mode

When LST (latest start time) interrupt occurs for \( T \):

- if \( \text{value}(T) > (1+\sqrt{k})(\text{value}(\text{current})+\text{value}(\text{preempted})) \)

then schedule \( T \) (and remove preempted status from all tasks)

else abandon \( T \)
Task Completion

- Schedule the task with earliest deadline if this task does not prevent completion of any recently-preempted task.

- If it does, then schedule the task with the earliest deadline among the recently-preempted tasks.
Proof Technique

Divide time into intervals.

- An interval ends when a task completes and there are no preempted tasks.

- An interval begins when a task is released to an idle system or at an interval endpoint.

The intervals partition the tasks.

First, prove that algorithm uses its time in each interval well.

Later, bound amount of value a clairvoyant algorithm can obtain with respect to tasks associated with that interval.
What Algorithm Achieves

$I$ — interval (by construction, no idle time)

\[
\text{achievedvalue}(I) = \text{ordervalue}(I) + \text{lstvalue}(I)
\]

- **lstvalue** — value of task (if any) that completed and was last scheduled by an Latest start time (LST) interrupt.

Fact: Once a task is scheduled by an lst interrupt, all other tasks will be so scheduled until one completes, ending the interval.

- **ordervalue** — value of tasks that completed and that were never scheduled by an lst interrupt.
Main Lemma

For every interval $I$, 

$$\text{length}(I) \leq \text{ordervalue}(I) + (1 + (1/\sqrt{k}))\text{lstvalue}(I)$$

This then gives us a lower bound on $\text{achievedvalue}(I)$ with respect to the time when the algorithm is busy.
Proof Idea

What can take time during the interval?

- Tasks can finish without being overloaded — all this time is accounted for in $\text{ordervalue}(I)$.

- Preempted tasks may take time, but never finish because of a successful 1st interrupt — the interrupting task ($T_1$) must have sufficient value to account for the value of those tasks and hence their time.

\[
\text{value}(T_1) > (1 + \sqrt{k})(\text{value(preempted)} + \text{value(current)})
\]
Proof Idea Continued

- Lst-scheduled tasks may take time, but never finish — Again, the reason must be that another task had \((1 + \sqrt{k})\) times the value of the unfinished task.

The net effect is that one won’t throw away a task unless the winning task has much more value.
Bounding the Clairvoyant Scheduler

• We have a lower bound on what the algorithm can achieve during the time when it executes.

• Now, we want to bound the value obtained by a clairvoyant scheduler.

• Overall Strategy: partition the tasks based on what D-Over does. Show that the clairvoyant can’t obtain too much value among the tasks that the algorithm doesn’t complete.
Task Partitioning

D-Over divides the tasks into three disjoint sets based on what it does with them:

- **poslax tasks** — tasks that complete with some time to spare.

- **zerolax tasks** — tasks that complete exactly on time.

- **failed tasks** — tasks that fail to complete, because they are abandoned at their 1st times.
Gifts to Clairvoyant Scheduler

We will do two favors for the clairvoyant scheduler that can only increase the value it obtains.

- Clairvoyant scheduler gets all the value of the *poslax* tasks without spending any time to get it.

- During any time when D-Over was busy (union of time in intervals), the clairvoyant scheduler can get value density $k$ by remaining idle.

  (Like a farm subsidy.)
Gift Lemma

Let $C(\text{SetOfTasks}) = \text{value clairvoyant algorithm can obtain on } \text{SetOfTasks if it has nothing else to do.}$

$$C(\text{failed } \cup \text{zerolax } \cup \text{ poslax}) \leq C(\text{failed } \cup \text{zerolax}) + C(\text{ poslax})$$

Proof: Each task has less competition.
Bounding Lemma

\[ C(\text{failed} \cup \text{zerolax}) \leq (k + 1 + \sqrt{k}) \cdot \text{achieved value} + \sqrt{k} \cdot \text{zerolax} \]

Basic argument:

- Tasks in zerolax can execute only within the busy period.

- Failed tasks can’t have too much value or D-Over would have executed them instead of abandoning them.
Main Theorem (Competitive Factor)

D-Over has a competitive factor of \( \frac{1}{(1+\sqrt{k})^2} \).

- \( \sqrt{k} \cdot \text{zeroval} \leq \sqrt{k} \cdot \text{achievedvalue} - \text{posval} \), by definition.

- \( C'(\text{failed} \cup \text{zerolax}) \leq (k + 1 + \sqrt{k}) \cdot \text{achievedvalue} + \sqrt{k} \cdot \text{achievedvalue} - \text{posval} \), from bounding lemma and above.
Main Theorem — continued

- $C'(\text{failed} \cup \text{zerolax} \cup \text{poslax})$
  $\leq$
  $C'(\text{failed} \cup \text{zerolax}) + C'(\text{poslax})$
  $\leq$
  $((1+\sqrt{k})^2 \cdot \text{achievedvalue} - \text{poslax}) + \text{poslax}$
  from gift lemma and simple manipulation.

  $= (1 + \sqrt{k})^2 \cdot \text{achievedvalue}$
Conclusion

- D-over does as well as any on-line scheduler can possibly do.

- Embodies many heuristics so should perform well in practice.

- Open problems:
  - parallel scheduling guarantees
  - tightness of some bounds
  - orthogonality of various weakenings of the model
Auctions  There are three basic kinds of auctions:

- **English**: bid up until there is only one person left.  
  (Winner pays one more than the second highest bidder’s limit.  Time is proportional to number of bid possibilities.)

- **Dutch**: price starts high and goes down until someone speaks up.  
  (Winner will bid his limit price, which is usually higher than the second highest bidder’s limit.  Much faster.)

- **Sealed**: send in your bid and highest one wins.  
  (Fast, but auctioneers may tell secrets.  In pure form, winner will bid his limit price.  Better is for winner to pay one more than second highest bidder.)
Anonymous Sealed Bids

- Anonymous for everyone but the winner.

- Uses modified sealed bid approach (one more than second highest bidder).

- Has $R (> 2)$ referees that don’t collude to discover bids.

- Any ideas?
Doug Tygar’s Auction

- Bidder b computes his privately bid, say n, and then generate a vector of values: r1, r2, ... rn, 0, ..., 0 0. where the total length k of this vector is the highest value that is conceivable. Surprising part: the ri’s are random non-zero values.

- Bidder b then splits this vector into 3 (R = 3) pieces:
  r11 r12 ... r1k
  r21 r22 .... r2k
  r31 r32 .... r3k

- such that adding the three together mod M (for some suitably large prime M) gives back the original vector.
Dispersal Step

- Each bidder sends vector 1 to auctioneer 1, vector 2 to auctioneer 2, and vector 3 to auctioneer 3.

- The auctioneers sum up the received vectors mod M. Each then generates a single vector of length k. They then exchange and add their vectors mod M.
Highest Two Bidders

• The last non-zero entry is the highest bid \( b \). Here’s why: all higher entries must be 0 since each bidder will have set that entry to 0 and addition mod \( M \) is associative and commutative.

• Since sealed auctions work best if the winning bidder in fact pays the amount offered by the second highest bidder, the algorithm goes on to find the second highest bid. One way to do this is to subtract away \( b \)'s vector from the total.
Financial Applications of Cryptography

- D. E. Shaw trades “baskets” of securities for institutional investors.

- There is an interesting zero-knowledge problem. If the investor tells a trading company what they want to do, the trading company has information it can use. Conversely, if the trading company tells the institutional investor its strategy for market buying and selling, the trading company loses its proprietary algorithm.

- How would you solve this?
D. E. Shaw solution

- put the software on the investor site. The software runs, perhaps after accessing market information. The software spits out a 6 letter code. The investor calls D. E. Shaw and D. E. Shaw tells the investor which price corresponds to this 6 letter code.

- The idea is that this makes it hard for the investor to reverse engineer the D. E. Shaw algorithm, while protecting the investor from divulging information to D. E. Shaw.
Cheap Security for Electronic Commerce

- The biggest challenge Ecommerce poses to security is to make purchases extremely cheap and secure.

- Viewing a picture or a short video should not cost much, nor should the security.

- Physical cash is clearly impractical. So, credit cards would seem to be the answer. But credit card transactions cost a few dollars per transaction, so the question is: how to do all this cheaply?
Payment Options

• Consider the most frequent act: a customer wants to pay some money to buy a digital product.

• Should be unforgeable.
  Should be transferable from customer to vendor.
  Should be spent only once.
  Should be private between customer and vendor.

• Three parties: customer, vendor, and brokers.

American Express Protocol
• Customer pays broker (AMEX) money and receives a one-time number. Customer then buys from vendor.

• Vendor verifies with Broker that the number is legitimate and for the right amount.
Fault Tolerance on Wall Street

- Memory fails, disks fail (in batches), fires happen (Credit Lyonnais, NY Stock Exchange), and power grids fail. If your system is still alive, you have a big advantage.

- You can even let your competitors use your facilities … for a price.
Case: Bond Futures

- Server for trading bond futures having to do with home mortgages.

- Application used only a few days per month, but the load is heavy. During a weekend batch run, 11 out of 12 two-gigabyte disks from a single vendor-batch failed.
High Availability Servers

• A pair of shared memory multiprocessors attached to RAID disks.

• If the primary multiprocessor fails, the backup does a warm start from the disks.

• If a disk fails, RAID masks it.

• Does not survive disasters or correlated failures.
Writes go to the primary and into the high availability disk subsystem. This subsystem is normally a RAID device, so can survive one or more disk failures. If the primary fails, the secondary works off the same disk image (warm start recovery).

Vulnerability: High availability disk subsystem fails entirely.
Dump and Load

- Full dump at night. Incremental dumps every three minutes.

- Can lose committed transactions, but there is usually a paper trail.

- Backup can be far away.
Replication Server

- Full dump nightly. All operations at the primary are sent to the secondary after commit on the primary.

- May lose a few seconds of committed transactions.

- Slight pain to administer, e.g. schemas, triggers, log size.
Basic architecture of a replication server.
The backup reads operations after they are committed on the primary. Upon failure, the secondary becomes the primary by changing the interface file configuration variables.

Vulnerability: if there is a failure of the primary after commit at the primary but before the data reaches the secondary, we have trouble.
Remote Mirroring

- Writes to local disks are mirrored to disks on a remote site. The commit at the local machine is delayed until the remote disks respond.

- Backup problems may cause primary to halt.

- Reliable buffering can be used (e.g. Qualix), but the net result is rep server without the ability to query the backup.
Two Phase Commit

- Commits are coordinated between the primary and backup.

- Blocking can occur if the transaction monitor fails. Delays occur if backup is slow.

- Wall Street is scared of this.
Two phase commit: transaction manager ensures that updates on the primary and secondary are commit-consistent. This ensures that the two sides are in synchrony.

Vulnerability: blocking or long delays may occur at the primary either due to delays at the secondary (in voting) or failure of the transaction manager.
Quorum Approach (e.g., DEC, HP, IBM, ISIS....)

- Servers are co-equal and are interconnected via a highly redundant wide area cable.

- Clients can be connected to any server. Servers coordinate via a distributed lock manager.

- Disks are connected with the servers at several points and to one another by a second wide area link.
Heartbeats

- Heartbeats monitor the connectivity among the various disks and processors.

- If a break is detected, one partition holding a majority of votes continues to execute.

- Any single failure of a processor, disk, site, or network is invisible to the end users (except for a loss in performance).
Quorum Approach as Used in most Stock and Currency Exchanges.
Survives Processor, Disk, and Site failures.

Quorum approach used in most exchanges.
Which to Use

- Stock exchanges use the quorum approach.

- Midoffice database servers often use dump and load or rep server. Symmetric approaches that may cause the primary to delay are too scary.

- Don’t buy batches from one vendor.