Distributed Systems

Fault Tolerance

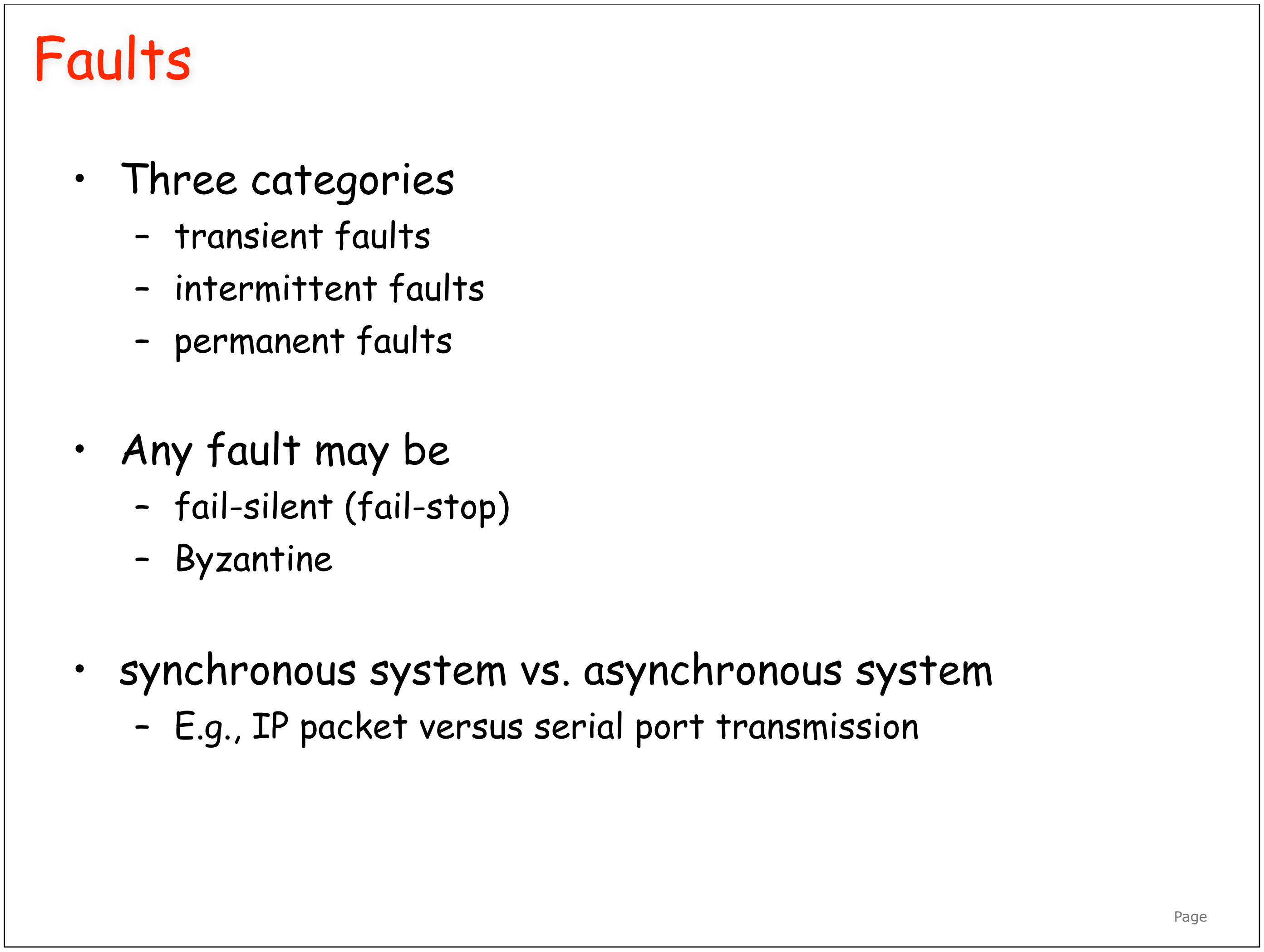
Paul Krzyzanowski

Faults

- Deviation from expected behavior
- Due to a variety of factors:
  - Hardware failure
  - Software bugs
  - Operator errors
  - Network errors/outages

A **fault** in a system is some deviation from the expected behavior of the system -- a malfunction.

Faults may be due to a variety of factors, including hardware, software, operator (user), and network errors.



Faults

- Three categories
  - transient faults
  - intermittent faults
  - permanent faults
- Any fault may be
  - fail-silent (fail-stop)
  - Byzantine
- synchronous system vs. asynchronous system
  - E.g., IP packet versus serial port transmission

Page

Faults can be classified into one of three categories:

transient faults: these occur once and then disappear. For example, a network message transmission times out but works fine when attempted a second time.

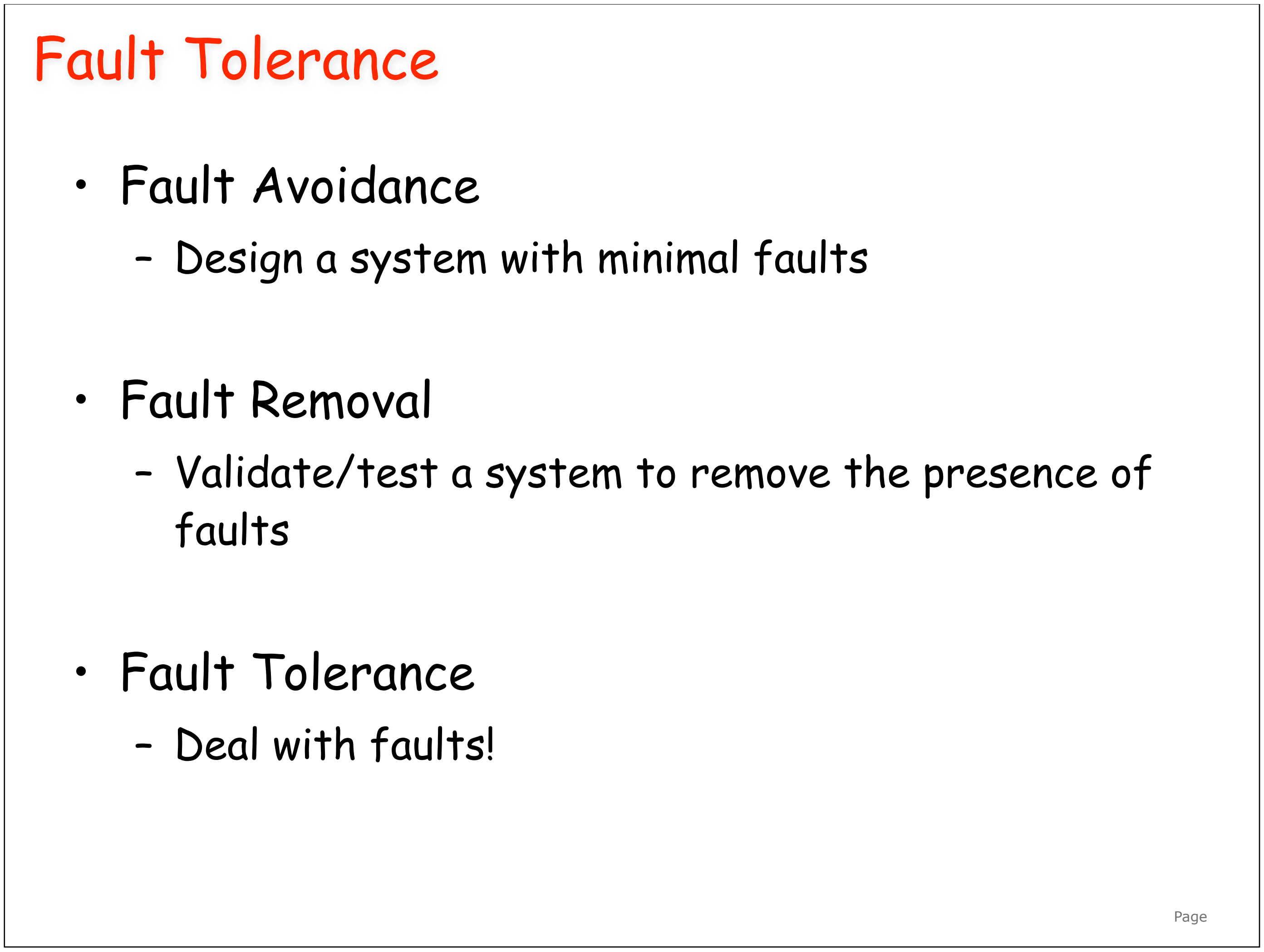
Intermittent faults: these are the most annoying of component faults. This fault is characterized by a fault occurring, then vanishing again, then occurring, …

An example of this kind of fault is a loose connection.

Permanent faults: this fault is persistent: it continues to exist until the faulty component is repaired or replaced. Examples of this fault are disk head crashes, software bugs, and burnt-out hardware.

Any of these faults may be either a **fail-silent failure** (also known as **fail-stop**) or a **Byzantine failure**. A fail-silent fault is one where the faulty unit stops functioning and produces no ill output (it produces no output or produces output to indicate failure). A Byzantine fault is one where the faulty unit continues to run but produces incorrect results. Byzantine faults are obviously more troublesome to deal with.

When we discuss fault tolerance, the familiar terms *synchronous* and *asynchronous* take on different meanings. By a **synchronous system**, we refer to one that responds to a message within a known, finite amount of time. An **asynchronous system**, does not.



Fault Tolerance

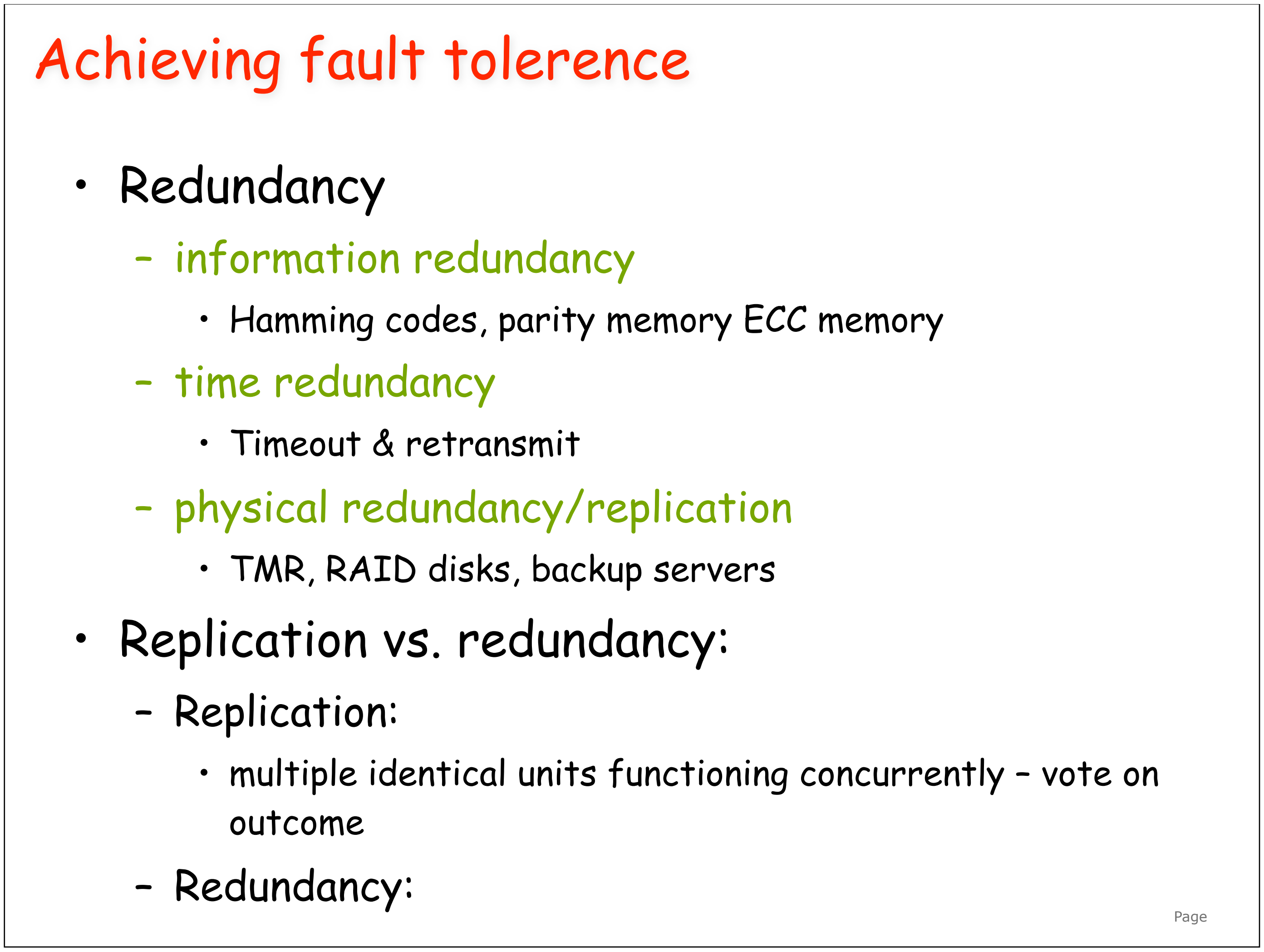
- Fault Avoidance
  - Design a system with minimal faults
- Fault Removal
  - Validate/test a system to remove the presence of faults
- Fault Tolerance
  - Deal with faults!

Page

We can try to design systems that minimize the presence of faults. Fault avoidance is where we go through design & validation steps to ensure that the system avoids being faulty in the first place.

Fault removal is an ex post facto approach where we encountered faults in the system and, through testing & verification, we manage to remove those faults (e.g., bug fixing, replacing failing components with better ones, adding better heat sinks, …).

Fault tolerance is the realization that we will have faults in our system (hardware and/or software) and we have to design the system in such a way that it will be tolerant of those faults. That is, it should compensate for the faults and continue to function.



Achieving fault tolerence

- Redundancy
  - information redundancy
    - Hamming codes, parity memory ECC memory
  - time redundancy
    - Timeout & retransmit
  - physical redundancy/replication
    - TMR, RAID disks, backup servers
- Replication vs. redundancy:
  - Replication:
    - multiple identical units functioning concurrently – vote on outcome
  - Redundancy:

Page

The general approach to building fault tolerant systems is **redundancy**.

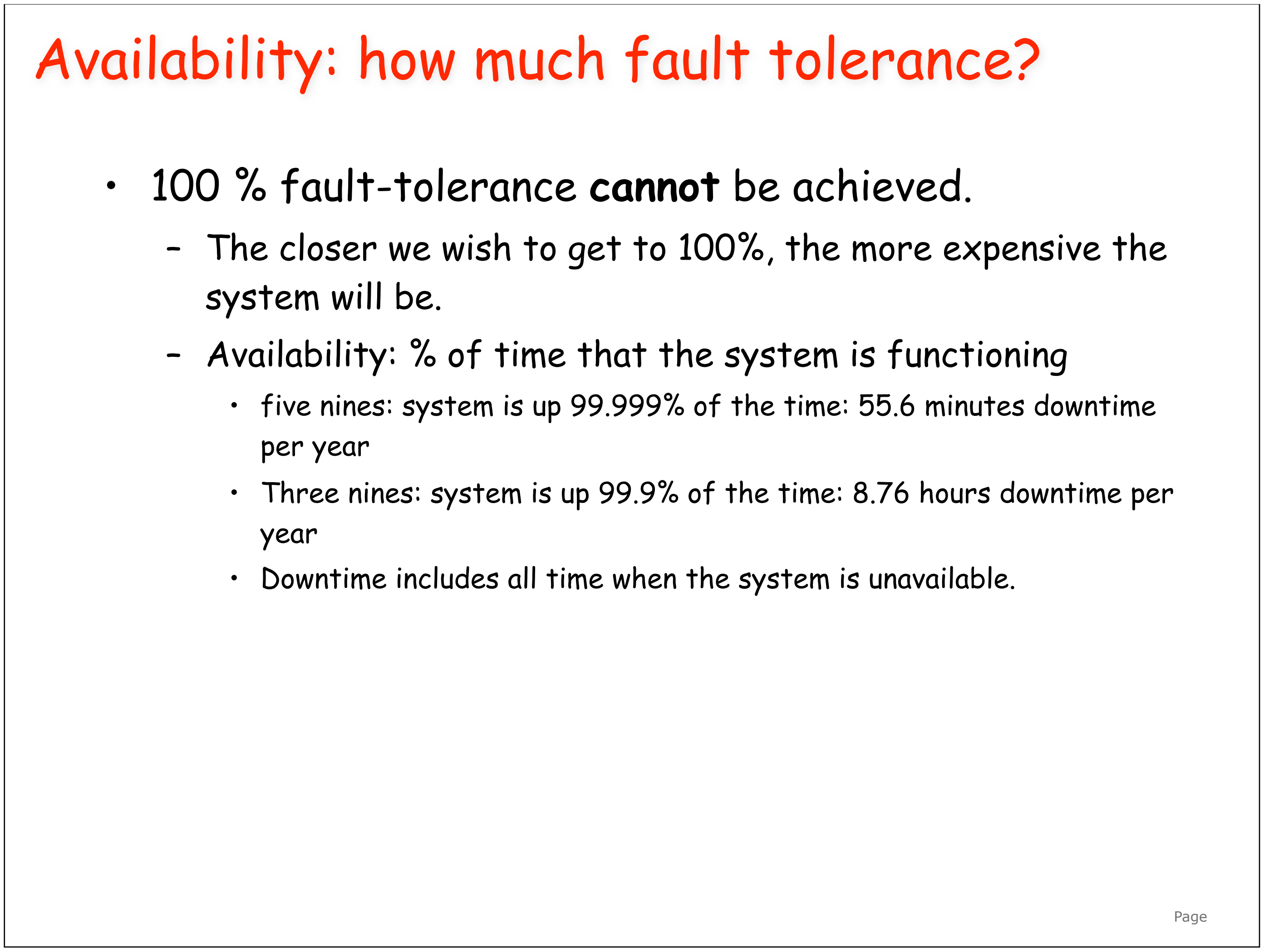
Redundancy may be applied at several levels.

Information redundancy seeks to provide fault tolerance through replicating or coding the data. For example, a Hamming code can provide extra bits in data to recover a certain ratio of failed bits. Sample uses of information redundancy are parity memory, ECC (Error Correcting Codes) memory, and ECC codes on data blocks.

Time redundancy achieves fault tolerance by performing an operation several times. Timeouts and retransmissions in reliable point-to-point and group communication are examples of time redundancy. This form of redundancy is useful in the presence of transient or intermittent faults. It is of no use with permanent faults. An example is TCP/IP's retransmission of packets.

Physical redundancy deals with devices, not data. We add extra equipment to enable the system to tolerate the loss of some failed components. RAID disks and backup name servers are examples of physical redundancy.

When addressing physical redundancy, we can differentiate *redundancy* from *replication*. With replication, we have several units operating concurrently and then a voting (quorum) system to select the outcome. With redundancy, one unit is functioning while others are available to fill in in case the unit ceases to work.



Availability: how much fault tolerance?

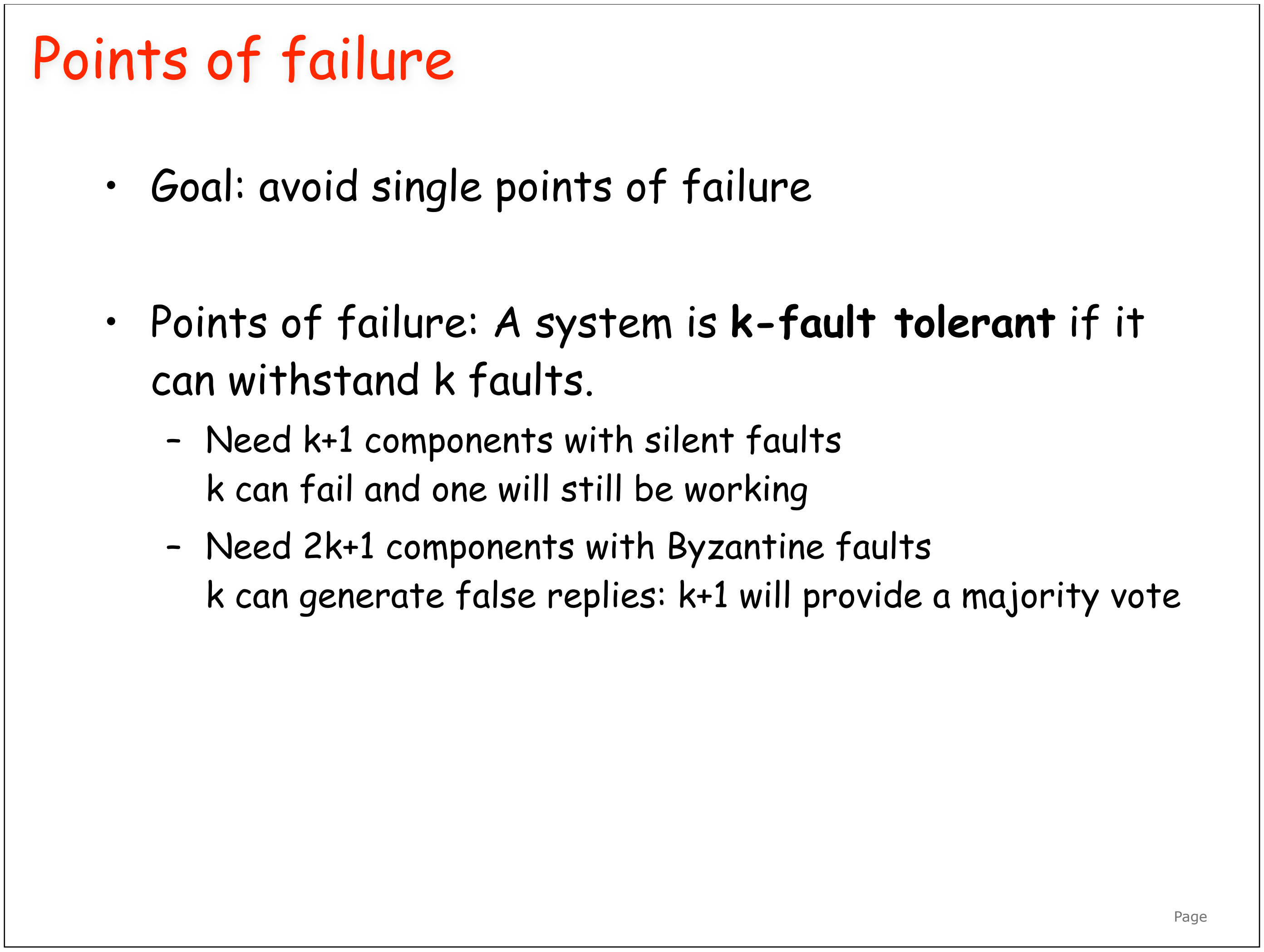
- 100 % fault-tolerance **cannot** be achieved.
  - The closer we wish to get to 100%, the more expensive the system will be.
  - Availability: % of time that the system is functioning
    - five nines: system is up 99.999% of the time: 55.6 minutes downtime per year
    - Three nines: system is up 99.9% of the time: 8.76 hours downtime per year
    - Downtime includes all time when the system is unavailable.

In designing a fault-tolerant system, we must realize that 100% fault tolerance can never be achieved. Moreover, the closer we with to get to 100%, the more costly our system will be.

To design a practical system, one must consider the degree of replication needed. This will be obtained from a statistical analysis for probable acceptable behavior. Factors that enter into this analysis are the average worst-case performance in a system without faults and the average worst-case performance in a system with faults.

Availability is typically measured by the percentage of time that a system is available to users. A system that's available 99.999% of the time (referred to as "five nines") will, on average, experience at most 55.6 minutes of downtime per year. This includes planned (hardware and software upgrades) and unplanned (network outages, hardware failures, fires, power outages, earthquakes) downtime.

Five nines is the classic standard of availability for telephony. It includes redundant processors, backup generators, and earthquake-resilient installation. If all that happens you your system is that you lose power for a day, your reliability is at 99.7%.



Points of failure

- Goal: avoid single points of failure
- Points of failure: A system is **k-fault tolerant** if it can withstand k faults.
  - Need k+1 components with silent faults
    k can fail and one will still be working
  - Need 2k+1 components with Byzantine faults
    k can generate false replies: k+1 will provide a majority vote

How much redundancy does a system need to achieve a given level of fault tolerance?

A system is said to be **k-fault tolerant** if it can withstand *k* faults.

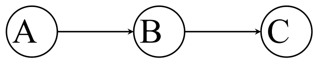
If the components fail silently, then it is sufficient to have *k+1* components to achieve *k* fault tolerance: *k* components can fail and one will still be working.

If the components exhibit Byzantine faults, hen a minimum of *2k+1* components are needed to achieve *k* fault tolerance. In the worst case, *k* components will fail (generating false results) and *k+1* components will remain working properly, providing a majority vote that is correct.

Active replication

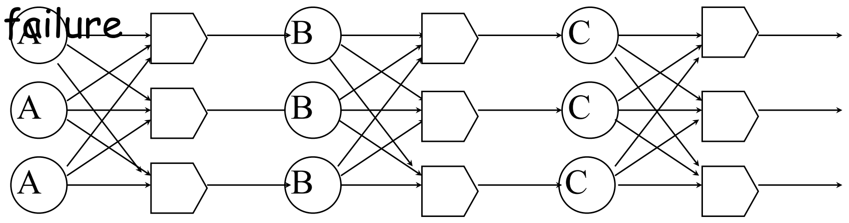
Technique for fault tolerance through physical redundancy

No redundancy:



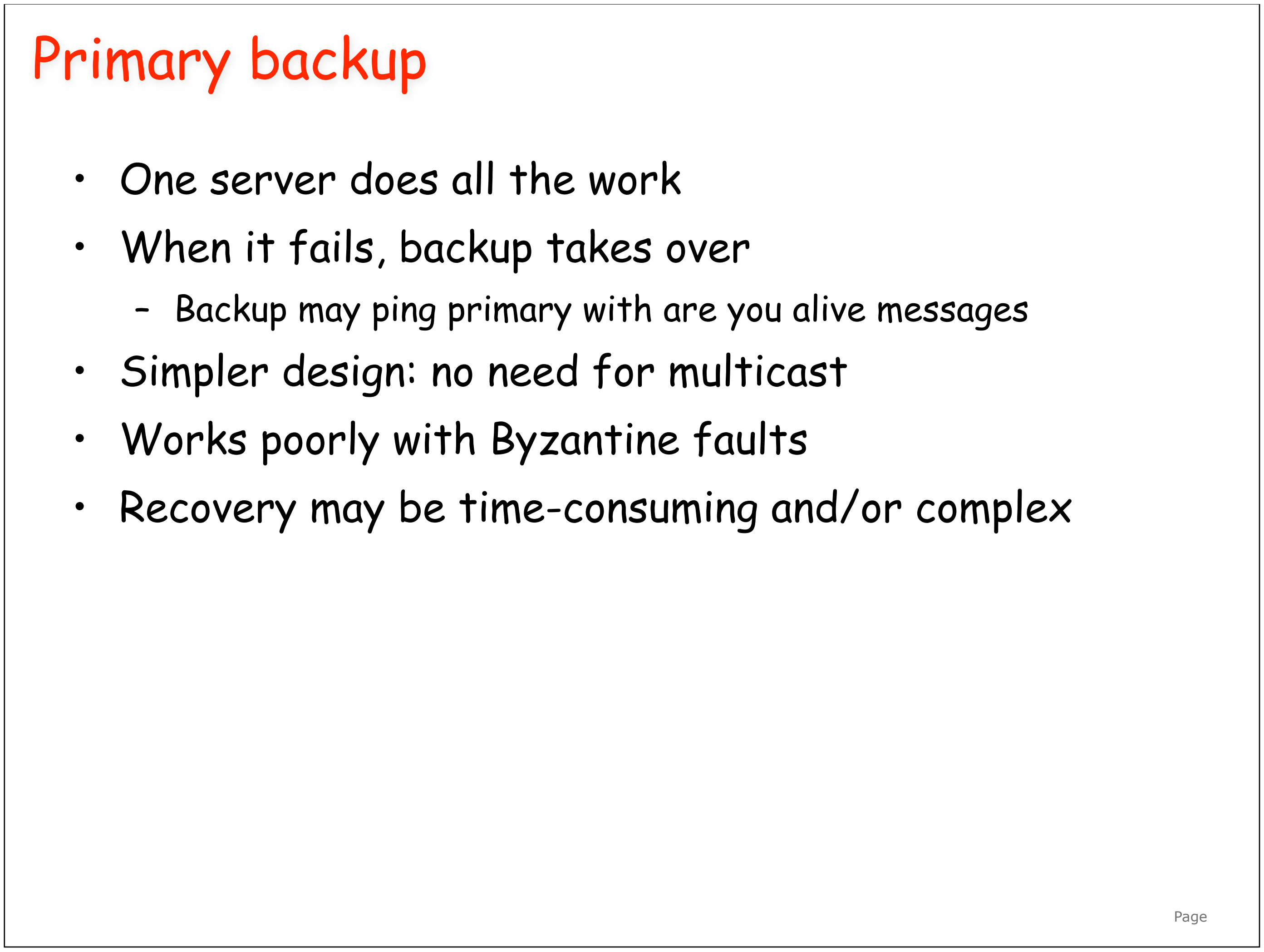
Triple Modular Redundancy (TMR):

Threefold component replication to detect and correct a single component failure



Active replication is a technique for achieving fault tolerance through physical redundancy. A common instantiation of this is **triple modular redundancy** (TMR). Under this system, we provide threefold replication of a component to detect and correct a single component failure. For example, instead of building a system where the output of A goes to the output of B, whose output goes to C (first figure), we replicate each component and place voters after each stage to pick the majority (second figure). The voter is responsible for picking the majority winner of the three inputs. The voters themselves are replicated because they too can malfunction.

In a software implementation, a client can replicate (or multicast) requests to each server. If requests are processed in order, all nonfaulty servers will yield the same replies. The requests must arrive reliably and in the same order on all servers. This requires the use of an atomic multicast.



Primary backup

- One server does all the work
- When it fails, backup takes over
  - Backup may ping primary with are you alive messages
- Simpler design: no need for multicast
- Works poorly with Byzantine faults
- Recovery may be time-consuming and/or complex

Page

To achieve fault tolerance with a primary backup, one server (the primary) does all the work.

When it fails, the backup takes over.

To find out whether a primary failed, a backup may send "*are you alive*" messages periodically. If the response times out, then the backup may assume that the primary failed and it will take over. If the system is asynchronous, there are no upper bounds on a timeout value. This is a problem. A possible solution is to use some hardware mechanism to detect and forcibly stop the primary.

This system is easier to design since requests do not have to go to the entire group, alleviating message ordering problems.

Fewer machines are needed – only a single backup server.

However, if the backup is put in service, another backup is needed *immediately*.

Backup servers work poorly with Byzantine faults, since the backup will be receiving erroneous data and not detecting it as such.

Recovery from a primary failure may be time-consuming and/or complex.

Under this scheme, messages should be reliably sent and each message should carry an identifier to avoid redoing the work when a backup takes over (if the request gets retransmitted).

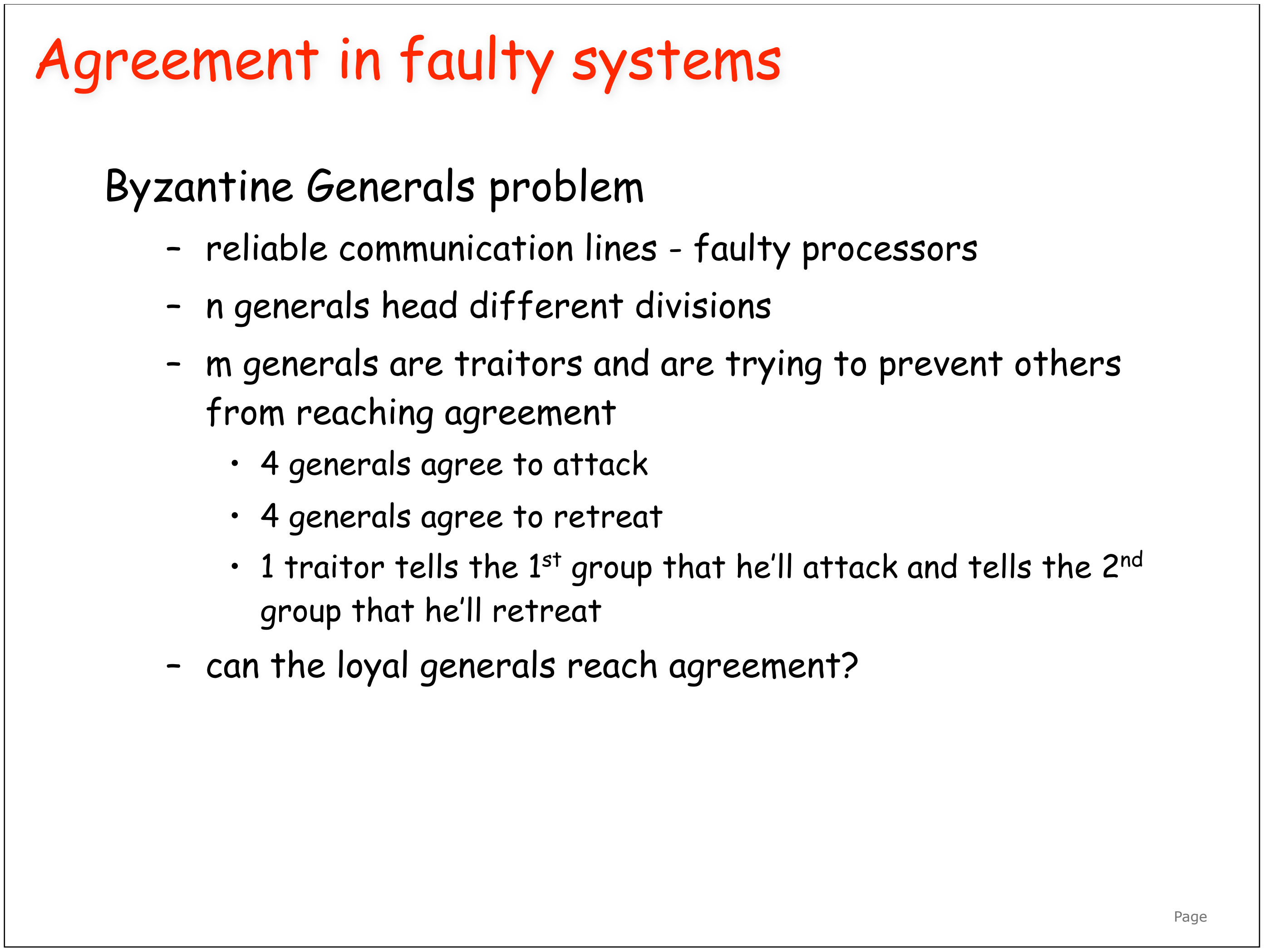
Agreement in faulty systems

Two army problem

- good processors - faulty communication lines
- coordinated attack
- multiple acknowledgement problem

Distributed processes often have to agree on something. For example, elect a coordinator, commit a transaction, divide tasks, coordinate a critical section, etc. What happens when the processes and/or the communication lines are imperfect?

We'll first examine the case of good processors but faulty communication lines. This is known as the **two army** problem and can be summarized as follows. Two divisions of an army, A and B, coordinate an attack on another army, C. A and B are physically separated and use a messenger to communicate. A sends a messenger to B with a message of *"let's attack at dawn"*. B receives the message and agrees, sending back the messenger with an *"OK"* message. The messenger arrives at A, but A realizes that B didn't know whether the messenger made it back safely. If B does is not convinced that A received the acknowledgement, then it will not be confident that the attack should take place. A may choose to send the messenger back to B with a message of *"A got the OK"* but A will then be unsure as to whether B received this message. This is also known as the *multiple acknowledgment* problem. It demonstrates that even with non-faulty processors, agreement between two processes is not possible with unreliable communication. The best we can do is hope that it usually works.



Agreement in faulty systems

Byzantine Generals problem

- reliable communication lines - faulty processors
- n generals head different divisions
- m generals are traitors and are trying to prevent others from reaching agreement
  - 4 generals agree to attack
  - 4 generals agree to retreat
  - 1 traitor tells the 1st group that he'll attack and tells the 2nd group that he'll retreat
- can the loyal generals reach agreement?

The other interesting case to consider is that of reliable communication lines but faulty processors. This is known as the **Byzantine Generals Problem**. There are *n* army generals who head different divisions. Communication is reliable (radio or telephone) but *m* of the generals are traitors (faulty) and are trying to prevent others from reaching agreement by feeding them incorrect information. The question is *can the loyal generals still reach agreement?* Specifically, each general knows the size of his division. At the end of the algorithm can each general know the troop strength of every other loyal division? Lamport demonstrated a solution that works for certain cases which is covered in Tannenbaum's text (pp. 221-222). The conclusion for this problem is that any solution to the problem of overcoming *m* traitors requires a minimum of $3m+1$ participants ($2m+1$ loyal generals). This means that more than 2/3 of the generals must be loyal. Moreover, it was demonstrated that no protocol can overcome *m* faults with fewer than $m+1$ rounds of message exchanges and $O(mn^2)$ messages. Clearly, this is a rather costly solution. While the Byzantine model may be applicable to certain types of special-purpose hardware, it will rarely be useful in general purpose distributed computing environments.

Agreement in faulty systems

Byzantine Generals problem

- Solutions require:
  - 3m+1 participants for m traitors (2m+1 loyal generals)
  - m+1 rounds of message exchanges
  - $O(m^2)$ messages
- Costly solution!

Example: ECC memory

- Memory chips designed with Hamming code logic
- Most implementations correct single bit errors in a memory location and detect multiple bit errors.
- Example of information redundancy

Example: Failover via DNS SRV

- Goal: allow multiple machines (with unique IP addresses in possibly different locations) to be represented by one hostname
- Instead of using DNS to resolve a hostname to one IP address, use DNS look up SRV records for that name.
  - Each record will have a priority, weight, and server name
  - Use the priority to pick one of several servers
  - Use the weight to pick servers of the same priority (for load balancing)
  - Then, once you picked a server, use DNS to look up its address

- Commonly used in voice-over-IP systems to pick a SIP server/proxy
- MX records (mail servers) take the same approach: use DNS to find several mail servers and pick one that works
- Example of physical redundancy

Example: RAID 1 (disk mirroring)

- RAID = redundant array of independent disks
- RAID 1: disk mirroring
  - All data that is written to one disk is also written to a second disk
  - A block of data can be read from either disk
  - If one disk goes out of service, the remaining disk will still have the data
- Example of physical redundancy

Example: RAID-4/RAID-5

- Block-level striping + parity
- Blocks are spread out across N disks and a parity block is written to disk N+1. The parity is the exclusive-or of the set of blocks in each stripe.
- If one disk fails, its contents are recovered by computing an exclusive-or of all the blocks in that stripe set together with the parity block
- RAID-5: same thing but the parity blocks are distributed among all the disks so that writing parity doesn't become a bottleneck.
- Example of information redundancy

Example: TCP retransmission

- Sender requires ack from a receiver for each packet
- If the ack is not received in a certain amount of time, the sender retransmits the packet

- Example of time redundancy

Transmission Control Protocol (TCP) relies that the sender receives an acknowledgement from the receiver whenever a packet is received.

If the sender does not receive that acknowledgment within a certain amount of time, it assumes that the packet was lost. The sender then retransmits the packet.

In Windows, the retransmission timer is initialized to three seconds. The default maximum number of retransmissions is five.

The retransmission time is adjusted based on the usual delay of a specific connection.

(See RFC 2581, TCP Congestion Control: http://www.ietf.org/rfc/rfc2581.txt)

The end