N-Version Programming

♦ N independent teams of programmers develop software to same specifications - N versions are run in parallel - output voted on
♦ If programs are developed independently - very unlikely that they will fail on same inputs
♦ Assumption - failures are statistically independent; probability of failure of an individual version = q
♦ Probability of no more than m failures out of N versions -

\[ p_{\text{ind}}(N, m, q) = \sum_{i=0}^{m} \binom{N}{i} q^i (1 - q)^{N-i} \]
Consistent Comparison Problem

- N-version programming is not simple to implement
- Even if all versions are correct - reaching a consensus is difficult
- Example:
  - \( V_1, \ldots, V_N \) - N independently written versions for computing a quantity \( X \) and comparing it to some constant \( C \)
  - \( X_i \) - value of \( X \) computed by version \( V_i \) \((i=1, \ldots, N)\)
  - The comparison with \( C \) is said to be consistent if either all \( X_i < c \) or all \( X_i \geq c \)

Consistency Requirement

- Example:
  - A function of pressure and temperature, \( f(p,t) \), is calculated
  - Action A1 is taken if \( f(p,t) < C \)
  - Action A2 is taken if \( f(p,t) \geq C \)
- Each version outputs action to be taken
- Ideally all versions consistent - output same action
- Versions are written independently - use different algorithms to compute \( f(p,t) \) - values will differ slightly
- Example: \( C=1.0000; N=3 \)
- All three versions operate correctly - output values: 0.9999, 0.9998, 1.0001
  - \( X_1, X_2 < C \) - recommended action is A1
  - \( X_3 > C \) - recommended action is A2
- Not consistent although all versions are correct
Consistency Problem

♦ Theorem: Any algorithm which guarantees that any two n-bit integers which differ by less than \(2^k\) will be mapped to the same m-bit output (where \(m+k \leq n\)), must be the trivial algorithm that maps every input to the same number.

♦ Proof:
  * We start with \(k=1\)
  * 0 and 1 differ by less than \(2^k\)
  * The algorithm will map both to the same number, say \(\alpha\)
  * Similarly, 1 and 2 differ by less than \(2^k\); so they will also be mapped to \(\alpha\)
  * Proceeding, we can show that 3,4,\ldots will all be mapped by this algorithm to \(\alpha\)
  * Therefore this is the trivial algorithm that maps all integers to the same number, \(\alpha\)

♦ Exercise: Show that a similar result holds for real numbers that differ even slightly from one another.

Consensus Comparison Problem

♦ If versions don’t agree – they may be faulty or not
♦ Multiple failed versions can produce identical wrong outputs due to correlated fault – system will select wrong output
♦ Can bypass the problem by having versions decide on a consensus value of the variable
♦ Before checking if \(X \geq C\), the versions agree on a value of \(X\) to use
♦ This adds the requirement: specify order of comparisons for multiple comparisons
♦ Can reduce version diversity, increasing potential for correlated failures
♦ Can also degrade performance – versions that complete early would have to wait
Another Approach - Confidence Signals

♦ Each version calculates $|X - C|$; if $< \delta$ for some given $\delta$, version announces low confidence in its output
♦ Voter gives lower weights to low confidence versions
♦ Problem: if a functional version has $|X - C| < \delta$, high chance that this will also be true of other versions, whose outputs will be devalued by voter
♦ The frequency of this problem arising, and length of time it lasts, depend on nature of application
♦ In applications where calculation depends only on latest inputs and not on past values - consensus problem may occur infrequently and go away quickly

Independent vs. Correlated Versions

♦ Correlated failures between versions can increase overall failure probability by orders of magnitude
♦ Example: $N=3$, can tolerate up to one failed version for any input; $q = 0.0001$ - an incorrect output once every ten thousand runs
♦ If versions stochastically independent - failure probability of 3-version system
  \[ q^3 + 3q^2(1 - q) \approx 3 \times 10^{-8} \]
♦ Suppose versions are statistically dependent and there is one fault, causing system failure, common to two versions, exercised once every million runs
♦ Failure probability of 3-version system increases to over $10^{-6}$, more than 30 times the failure probability of uncorrelated system
**Version Correlation Model**

- **Input space divided to regions:** different probability of input from region to cause a version to fail
- **Example:** Algorithm may have numerical instability in an input subspace - failure rate greater than average
- **Assumption:** Versions are stochastically independent in each given subspace $S_i$ -
  \[ \text{Prob}\{\text{both } V_1 \text{ and } V_2 \text{ fail } | \text{ input from } S_i\} = \text{Prob}\{V_1 \text{ fails } | \text{ input from } S_i\} \times \text{Prob}\{V_2 \text{ fails } | \text{ input from } S_i\} \]
- **Unconditional probability of failure of a version**
  \[ \text{Prob}\{V_1 \text{ fails}\} = \sum \text{Prob}\{V_1 \text{ fails } | \text{ input from } S_i\} \times \text{Prob}\{\text{input from } S_i\} \]
- **Unconditional probability that both fail**
  \[ \text{Prob}\{V_1 \text{ and } V_2 \text{ fail}\} = \sum \text{Prob}\{V_1 \text{ and } V_2 \text{ fail } | \text{ input from } S_i\} \times \text{Prob}\{\text{input from } S_i\} \]

**Version Correlation: Example 1**

- **Two input subspaces $S_1, S_2$ - probability 0.5 each**
- **Conditional failure probabilities:**
  \[
  \begin{array}{c|cc}
    & S_1 & S_2 \\
  \hline
  V_1 & 0.01 & 0.001 \\
  V_2 & 0.02 & 0.003 \\
  \end{array}
  \]
- **Unconditional failure probabilities:**
  \[
  \begin{align*}
  \text{P(V1 fails)} &= 0.01 \times 0.5 + 0.001 \times 0.5 = 0.0055 \\
  \text{P(V2 fails)} &= 0.02 \times 0.5 + 0.003 \times 0.5 = 0.0115 \\
  \end{align*}
  \]
- **If versions were independent, probability of both failing for same input = 0.0055x0.0115 = 6.33 × 10^{-5}**
- **Actual joint failure probability is higher**
  \[
  \text{P(V1 \& V2 fail)} = 0.01 \times 0.02 \times 0.5 + 0.001 \times 0.003 \times 0.5 = 1.02 \times 10^{-4} \\
  \]
- **The two versions are positively correlated:** both are more prone to failure in $S_1$ than in $S_2$
Version Correlation: Example 2

- **Conditional failure probabilities:**
  
<table>
<thead>
<tr>
<th>Version</th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>0.010</td>
<td>0.001</td>
</tr>
<tr>
<td>V2</td>
<td>0.003</td>
<td>0.020</td>
</tr>
</tbody>
</table>

- **Unconditional failure probabilities** - same as Example 1

- **Joint failure probability** -
  
  \[
P(V1 \& V2 \text{ fail}) = 0.01 \times 0.003 \times 0.5 + 0.001 \times 0.02 \times 0.5 = 2.5 \times 10^{-5}\]

- Much less than the previous joint probability or the product of individual probabilities

- **Tendencies to failure are negatively correlated:**

- V1 is better in S1 than in S2, opposite for V2 - V1 and V2 make up for each other's deficiencies

- **Ideally** - multiple versions negatively correlated

- **In practice** - positive correlation - since versions are solving the same problem

Causes of Version Correlation

- **Common specifications** - errors in specifications will propagate to software

- **Intrinsic difficulty of problem** - algorithms may be more difficult to implement for some inputs, causing faults triggered by same inputs

- **Common algorithms** - algorithm itself may contain instabilities in certain regions of input space - different versions have instabilities in same region

- **Cultural factors** - Programmers make similar mistakes in interpreting ambiguous specifications

- **Common software and hardware platforms** - if same hardware, operating system, and compiler are used - their faults can trigger a correlated failure
Achieving Version Independence - Incidental Diversity

◆ Forcing developers of different modules to work independently of one another
◆ Teams working on different modules are forbidden to directly communicate
◆ Questions regarding ambiguities in specifications or any other issue have to be addressed to some central authority who makes any necessary corrections and updates all teams
◆ Inspection of software carefully coordinated so that inspectors of one version do not leak information about another version

Achieving Version Independence - Methods for Forced Diversity

◆ Diverse specifications
◆ Diverse hardware and operating systems
◆ Diverse development tools and compilers
◆ Diverse programming languages
◆ Versions with differing capabilities

Diverse Specifications

◆ Most software failures due to requirements specification
◆ Diversity can begin at specification stage - specifications may be expressed in different formalisms
◆ Specification errors will not coincide across versions - each specification will trigger a different implementation fault profile
Diverse Hardware and Operating Systems
♦ Output depends on interaction between application software and its platform – OS and processor
♦ Both processors and operating systems are notorious for the bugs they contain
♦ A good idea to complement software design diversity with hardware and OS diversity - running each version on a different processor type and OS

Diverse Development Tools and Compilers
♦ May make possible "notational diversity" reducing extent of positive correlation between failures
♦ Diverse tools and compilers (may be faulty) for different versions may allow for greater reliability

Diverse Programming Languages
♦ Programming language affects software quality
♦ Examples:
  - Assembler - more error-prone than a higher-level language
  - Nature of errors different - in C programs - easy to overflow allocated memory - impossible in a language that strictly manages memory
  - No faulty use of pointers in Fortran - has no pointers
  - Lisp is a more natural language for some artificial intelligence (AI) algorithms than are C or Fortran
♦ Diverse programming languages may have diverse libraries and compilers - will have uncorrelated (or even better, negatively-correlated) failures
Choice of Programming Language

♦ Should all versions use best language for problem or some versions be in other less suited languages?
  * If same language - lower individual fault rate but positively correlated failures
  * If different languages - individual fault rates may be greater, but overall failure rate of N-version system may be smaller if less correlated failures
  * Tradeoff difficult to resolve - no analytical model exists - extensive experimental work is necessary

Versions With Differing Capabilities

♦ Example: One rudimentary version providing less accurate but still acceptable output
♦ 2nd simpler, less fault-prone and more robust
♦ If the two do not agree - a 3rd version can help determine which is correct
♦ If 3rd very simple, formal methods may be used to prove correctness

Back-to-Back Testing

♦ Comparing intermediate variables or outputs for same input - identify non-coincident faults

♦ Intermediate variables provide increased observability into behavior of programs
♦ But, defining intermediate variables constrains developers to producing these variables - reduces program diversity and independence
Single Version vs. N Versions

♦ Assumption: developing N versions - N times as expensive as developing a single version
♦ Some parts of development process may be common, e.g. - if all versions use same specifications, only one set needs to be developed
♦ Management of an N-version project imposes additional overheads
♦ Costs can be reduced - identify most critical portions of code and only develop versions for these
♦ Given a total time and money budget - two choices:
  * (a) develop a single version using the entire budget
  * (b) develop N versions
♦ No good model exists to choose between the two

Experimental Results

♦ Few experimental studies of effectiveness of N-version programming
♦ Published results only for work in universities
♦ One study at the Universities of Virginia and California at Irvine
  * 27 students wrote code for anti-missile application
  * Some had no prior industrial experience while others over ten years
  * All versions written in Pascal
  * 93 correlated faults identified by standard statistical hypothesis-testing methods: if versions had been stochastically independent, we would expect no more than 5
  * No correlation observed between quality of programs produced and experience of programmer
Recovery Block Approach

♦ N versions, one running - if it fails, execution is switched to a backup
♦ Example - primary + 3 secondary versions
♦ Primary executed - output passed to acceptance test
♦ If output is not accepted - system state is rolled back and secondary 1 starts, and so on
♦ If all fail - computation fails
♦ Success of recovery block approach depends on failure independence of different versions and quality of acceptance test

Recovery Block Approach - Analytical Model

♦ Assumption - different versions fail independently
♦ Notations:
  ♦ E - the event - output of a version is erroneous
  ♦ T - the event - test fails (test detects a fault)
  ♦ f - failure probability of a version \( f = P(E) \)
  ♦ s - test sensitivity \( s = P(T|E) \)
  ♦ \( \sigma \) - test specificity \( \sigma = P(E|T) \)
  ♦ n - number of software versions
Success Probability

- **Scheme success:** success at stage \( i, 1 \leq i \leq n \) - test must fail at stages \( 1, \ldots, i-1 \) and at stage \( i \) software version is correct and output passes test

\[
\text{Prob[Success in stage } i \} = [P(T)]^{i-1} P(\bar{E} \cap \bar{T})
\]

\[
\text{Prob[Scheme is successful]} = \sum_{i=1}^{n} [P(T)]^{i-1} P(\bar{E} \cap \bar{T})
\]

\[
P(E \cap T) = P(T|E)P(E) = sf
\]

\[
P(T) = \frac{P(E \cap T)}{P(E)} - \frac{sf}{\sigma}
\]

\[
P(E|T) = 1 - P(E|T) = 1 - \sigma
\]

\[
P(E \cap T) = P(E|T)P(T) = (1 - \sigma) \frac{sf}{\sigma}
\]

\[
P(E) = 1 - P(E) = 1 - f
\]

\[
P(\bar{E} \cap \bar{T}) = P(\bar{E}) - P(E \cap T) = (1 - f) - (1 - \sigma) \frac{sf}{\sigma}
\]

Success Probability as a function of \( f \)

\[
\text{Prob[Scheme is successful]} = \sum_{i=1}^{n} \left[ \frac{sf}{\sigma} \right]^{i-1} \left[ (1 - f) - (1 - \sigma) \frac{sf}{\sigma} \right]
\]

\[
= \frac{1 - \left( \frac{sf}{\sigma} \right)^n}{1 - \frac{sf}{\sigma}} \left[ (1 - f) - (1 - \sigma) \frac{sf}{\sigma} \right]
\]

Example -

- \( n=3 \), 2 values of \( s \) and \( \sigma \)
Distributed Recovery Blocks

♦ Two nodes carry identical copies of primary and secondary
♦ Node 1 executes the primary - in parallel, node 2 executes the secondary
♦ If node 1 fails the acceptance test, output of node 2 is used (provided that it passes the test)
♦ Output of node 2 can also be used if node 1 fails to produce an output within a prespecified time

Distributed Recovery Blocks - cont.

♦ Once primary fails, roles of primary and secondary are reversed
♦ Node 2 continues to execute the secondary copy, which is now treated as primary
♦ Execution by node 1 of primary is used as a backup
♦ This continues until execution by node 2 is flagged erroneous, then system toggles back to using execution by node 2 as a backup
♦ Rollback is not necessary - saves time - useful for real-time system with tight task deadlines
♦ Scheme can be extended to N versions (primary plus N-1 secondaries run in parallel on N processors
Exception Handling

- Exception - something happened during execution that needs attention
- Control transferred to exception-handler-routine
- Example: \( y = a \times b \), if overflow - signal an exception
- Effective exception-handling can make a significant improvement to system fault tolerance
- Over half of code lines in many programs devoted to exception-handling
- Exceptions deal with
  - (a) domain or range failure
  - (b) out-of-ordinary event (not failure) needing special attention
  - (c) timing failure

Domain and Range Failure

- Domain failure - illegal input is used
- Example: if \( X, Y \) are real numbers and \( X = \sqrt{Y} \) is attempted with \( Y = -1 \), a domain failure occurs
- Range failure - program produces an output or carries out an operation that is seen to be incorrect in some way
- Examples include:
  - Encountering an end-of-file while reading data from file
  - Producing a result that violates an acceptance test
  - Trying to print a line that is too long
  - Generating an arithmetic overflow or underflow
Out-of-the-Ordinary Events

♦ Exceptions can be used to ensure special handling of rare, but perfectly normal, events

♦ Example - Reading the last item of a list from a file - may trigger an exception to notify invoker that this was the last item

♦ Timing Failures:
  ♦ In real-time applications, tasks have deadlines
  ♦ If deadlines are violated - can trigger an exception
  ♦ Exception-handler decides what to do in response: for example - may switch to a backup routine

Requirements of Exception-Handlers

♦ (1) Should be easy to program and use
  ♦ Be modular and separable from rest of software
  ♦ Not be mixed with other lines of code in a routine - would be hard to understand, debug, and modify

♦ (2) Exception-handling should not impose a substantial overhead on normal functioning of system

♦ Exceptions be invoked only in exceptional circumstances

♦ Exception-handling not inflict a burden in the usual case with no exception conditions

♦ (3) Exception-handling must not compromise system state - not render it inconsistent
Software Reliability Models

♦ Software is often the major cause of system unreliability - accurately predicting software reliability is very important
♦ Relatively young and often controversial area
♦ Many analytical models, some with contradictory results
♦ We describe three models - very small part of available models
♦ Not enough evidence to select the correct model
♦ Although models attempt to provide numerical reliability, they should be used mainly for determining software quality

Software Reliability - Definitions

♦ Defect (bug) - exists in the software when written
♦ Error - a deviation of the program operation from its exact requirements (as the result of the bug)
♦ Bugs exist in software once it is written; errors occur only when program is running (or is being tested)
♦ Software does not deteriorate with time like hardware - reliability remains constant if no changes are made
♦ Once an error occurs, the bug causing it is corrected; other bugs still remain; reliability will increase
♦ An accepted definition of software reliability is probability of error-free operation of a computer program in a specified environment for a specified time
Software Error Rate

♦ Software reliability models attempt to predict the error rate, \( \lambda(t) \), as a function of number of bugs in the software at time \( t \).

♦ \( \lambda(t) \) can be used to determine the length of testing time (and bug correction) required.

♦ When \( \lambda(t) \) - the predicted future error rate - goes below some threshold, the software can be released.

♦ Assumptions for all three models:
  * Software has initially some unknown number of bugs.
  * It is tested for a period of time, during which some of the bugs cause errors.
  * Whenever an error occurs, the bug causing it is fixed (fixing time negligible) without causing any additional bugs, thus reducing number of existing bugs by one.

♦ The models differ in their modeling of \( \lambda(t) \), and consequently, in the software reliability prediction.

Jelinski-Moranda Model - Assumptions

♦ At time 0 - software has a fixed (finite) number \( N(0) \) of bugs.

♦ At time \( t \), \( N(t) \) bugs remain.

♦ Error process is a non-homogeneous Poisson process with a rate \( \lambda(t) \) that may vary with time.

♦ \( \lambda(t) \) is proportional to \( N(t) \) : \( \lambda(t) = cN(t) \) for some \( c \).

♦ \( \lambda(t) \) is a step function:
  * Initial value \( \lambda_0 = \lambda(0) = cN(0) \).
  * Decreases by \( c \) whenever an error occurs and the bug causing it is corrected.
  * Is constant between errors.

♦ The (testing, not including fixing) time between errors (say, \( i \) and \( i+1 \)) is exponentially distributed with parameter \( \lambda(t) \) (\( t \) - the time of the \( ith \) error).

Page 17
Jelinski-Moranda Model – Reliability Calculation

♦ \( R(t) \) - probability of error-free operation during \([0, t]\)

\[
R(t) = e^{-\lambda_0 t}
\]

♦ Given an error occurred at time \( \tau \) - conditional future reliability = conditional probability that the interval \([\tau, \tau + t]\) will be error-free is

\[
R(t \mid \tau) = e^{-\lambda(\tau) t}
\]

♦ With testing, more bugs are caught and corrected, error rate decreases, future reliability increases

♦ Model assumes all bugs contribute equally to error rate, as expressed by the constant \( c \)

♦ Actually, some bugs are exercised more often, and the more difficult bugs to catch during testing are those that are not exercised often

Littlewood-Verrall Model - Assumptions

♦ \( N(0) \) initial bugs; \( N(t) \) bugs remain at time \( t \)

♦ \( M(t) = N(0) - N(t) \) - number of bugs discovered and corrected during \([0, t]\); \( M(0) = 0 \)

♦ The errors occur according to a nonhomogeneous Poisson process with rate \( \lambda(t) \)

♦ \( \lambda(t) \) is a random variable with a Gamma density function - two parameters \( \alpha \) and \( \Psi \)

♦ \( \Psi \) is a monotonically increasing function of \( M(t) \)

\[
f_{\lambda(t)}(\ell) = \frac{[\Psi(M(t))]^\alpha \ell^{\alpha-1} e^{-\Psi(M(t)) \ell}}{\Gamma(\alpha)}
\]

Where

\[
\Gamma(x) = \int_0^\infty e^{-y}y^{x-1}dy
\]
Littlewood-Verrall Model – Reliability

- The Gamma density function is easy to analyze and very flexible.

- After integrating $e^{-lt}$ with respect to $f_{\lambda(t)}(l)$ with $\Psi = \Psi(M(\tau)) = \Psi(0)$ we obtain

$$R(t) = \left(1 + \frac{t}{\Psi(0)}\right)^{-\alpha}$$

- A similar integration using $\Psi = \Psi(M(\tau))$ yields

$$R(t \mid \tau) = \left(1 + \frac{t}{\Psi(M(\tau))}\right)^{-\alpha}$$

Musa-Okumoto Model – Assumptions

- More widely used software reliability model.

- A very large number of initial bugs in the software.

- $M(t)$ - number of bugs discovered and corrected during time $[0,t]$.

- Failure rate after testing for time $t$ is

$$\lambda(t) = \lambda_0 e^{-c \mu(t)}$$

  * $\lambda_0$ - initial value of failure rate
  * $c$ - constant
  * $\mu(t) = E(M(t))$ - expected value of $M(t)$

- Intuitive basis for this model:

  * When testing starts, “easiest” bugs are caught quickly.
  * Remaining bugs are more difficult to catch.
  * The rate of errors drops exponentially as testing proceeds.
Musa-Okumoto Model - Reliability

- From $\lambda(t) = \lambda_0 e^{-ct}$ we get the differential equation
  $$\frac{d\mu(t)}{dt} = \lambda(t) = \lambda_0 e^{-ct}$$
- Whose solution is $\mu(t) = \frac{\ln(\lambda_0 ct + 1)}{c}$; $\lambda(t) = \frac{\lambda_0}{\lambda_0 ct + 1}$
- The resulting reliability is
  $$R(t) = e^{-\int_0^t \lambda(z) \, dz} = e^{-\mu(t)} = (1 + \lambda_0 ct)^{-\frac{1}{c}}$$
- And the future conditional reliability is
  $$R(t | \tau) = e^{-\int_{\tau}^{t+\tau} \lambda(z) \, dz} = e^{-\mu(t+\tau)} = \left(1 + \frac{\lambda_0 ct}{t + \lambda_0 c \tau}\right)^{-\frac{1}{c}}$$

Musa-Okumoto Model - Error Rates

- Very slow decay of failure rate - requiring significant amount of testing

![Graphs showing failure rate decay](image-url)
Selecting Model and Parameter Estimation

♦ (1) Which model is appropriate
♦ (2) How to estimate model parameters
♦ No comprehensive experimental data to guide users
♦ Study failure rate as a function of testing, and guess which model it follows
♦ Then - estimate its parameters
♦ Use standard statistical estimation techniques such as Maximum Likelihood and Least Squares methods

Fault-Tolerant Remote Procedure Call (RPC)
♦ A mechanism by which one process can call another process executing on some other processor - widely used in distributed computing
♦ Two ways of making RPCs fault tolerant - we assume that processes are fail-stop.
♦ (1) Primary-Backup Approach:
♦ Each process is implemented as primary and backup processes, running on separate nodes
♦ RPCs are sent to both copies - only the primary executes them
♦ If primary fails - secondary activated and completes execution
♦ RPCs can be
  * retryable - can be executed multiple times without violating correctness
  * nonretryable - can be completed exactly once
The Circus Approach

- Client & server processes replicated
  * Replicate sets - troupes
- Example: 4 replicates of client, make identical calls to 4 server replicates
- Each call has a sequence number
- A server waits for all 4 identical calls before executing the RPC
- Results are then sent back to each client - marked by a sequence number to uniquely identify them
- Client waits until receiving identical replies from each server before accepting input (subject to a timeout - prevent from waiting forever for a failed server process)
- Alternative - take the first reply and ignore the rest
- Additional complication: Multiple client troupes can send concurrent calls to same server troupe - each member of server troupe must serve calls in exactly the same order

Optimistic & Pessimistic Approaches

- Optimistic approach - no special attempt to ensure preservation of order - performs poorly if ordering is often not preserved
- Pessimistic approach - built-in mechanisms for preserving order
- Simple optimistic scheme:
  - Each member of server troupe receives requests from one or more client troupes
  - When it completes processing it sends ready_to_commit message to each element of client troupe
  - Waits until every member of client troupe acknowledges this call, before proceeding to commit
  - Similarly on client side: waits until it receives ready_to_commit from every member of server troupe, before ACK the call
  - Once server receives ACK from each member of client troupe, it commits
  - This approach ensures correct functioning by forcing deadlock if the serial order is violated
Simple Optimistic Scheme - Example

- Two client troupes C1 & C2 making concurrent RPCs ρ1 and ρ2 to a server troupe consisting of servers S1 & S2
- If S1 tries to commit ρ1 first and then ρ2, while S2 works in the opposite order
- Once S1 is ready to commit ρ1, it sends a ready_to_commit to each member of C1, and waits to receive an ACK from each
- Similarly, S2 gets ready to commit ρ2, and sends a ready_to_commit to each member of C2
- Members of each client troupe will wait until hearing a ready_to_commit from both S1 and S2
- Since members of C1 will not hear from S2 and members of C2 will not hear from S1 there is a deadlock
  - Algorithms exist to detect such deadlocks in distributed systems
- Once the deadlock is detected, the operations can be aborted before being committed, and then retried