Надежность в распределенных системах

Отказы в распределенных системах

- Отказом системы называется поведение системы, не удовлетворяющее ее спецификациям. Последствия отказа могут быть различными.
- Отказ системы может быть вызван отказом (неверным срабатыванием) каких-то ее компонентов (процессор, память, устройства ввода/вывода, линии связи, или программное обеспечение).
- Отказ компонента может быть вызван ошибками при конструировании, при производстве или программировании. Он может быть также вызван физическим повреждением, изнашиванием оборудования, некорректными входными данными, ошибками оператора, и многими другими причинами.

Отказы в распределенных системах

- Отказы могут быть случайными, периодическими или постоянными.
- Случайные отказы (сбои) при повторении операции исчезают.
- Причиной такого сбоя может служить, например, электромагнитная помеха от проезжающего мимо трамвая. Другой пример - редкая ситуация в последовательности обращений к операционной системе от разных задач.
- Периодические отказы повторяются часто в течение какого-то времени, а затем могут долго не происходить. Примеры - плохой контакт, некорректная работа ОС после обработки аварийного завершения задачи.
- Постоянные (устойчивые) отказы не прекращаются до устранения их причины - разрушения диска, выхода из строя микросхемы или ошибки в программе.

Отказы в распределенных системах

- Отказы по характеру своего проявления подразделяются на «византийские» (система активна и может проявлять себя по-разному, даже злонамеренно) и «пропажа признаков жизни» (частичная или полная). Первые распознать гораздо сложнее, чем вторые. Свое название они получили по имени Византийской империи (330-1453 гг.), где расцветали конспирация, интриги и обман.
- Для обеспечения надежного решения задач в условиях отказов системы применяются два принципиально различающихся подхода - восстановление решения после отказа системы (или ее компонента) и предотвращение отказа системы (отказоустойчивость).

Восстановление после отказа

- Восстановление может быть прямым (без возврата к прошлому состоянию) и возвратное.
- Прямое восстановление основано на своевременном обнаружении сбоя и ликвидации его последствий путем приведения некорректного состояния системы в корректное. Такое восстановление возможно только для определенного набора заранее предусмотренных сбоев.
- При возвратном восстановлении происходит возврат процесса (или системы) из некорректного состояния в некоторое из предшествующих корректных состояний.

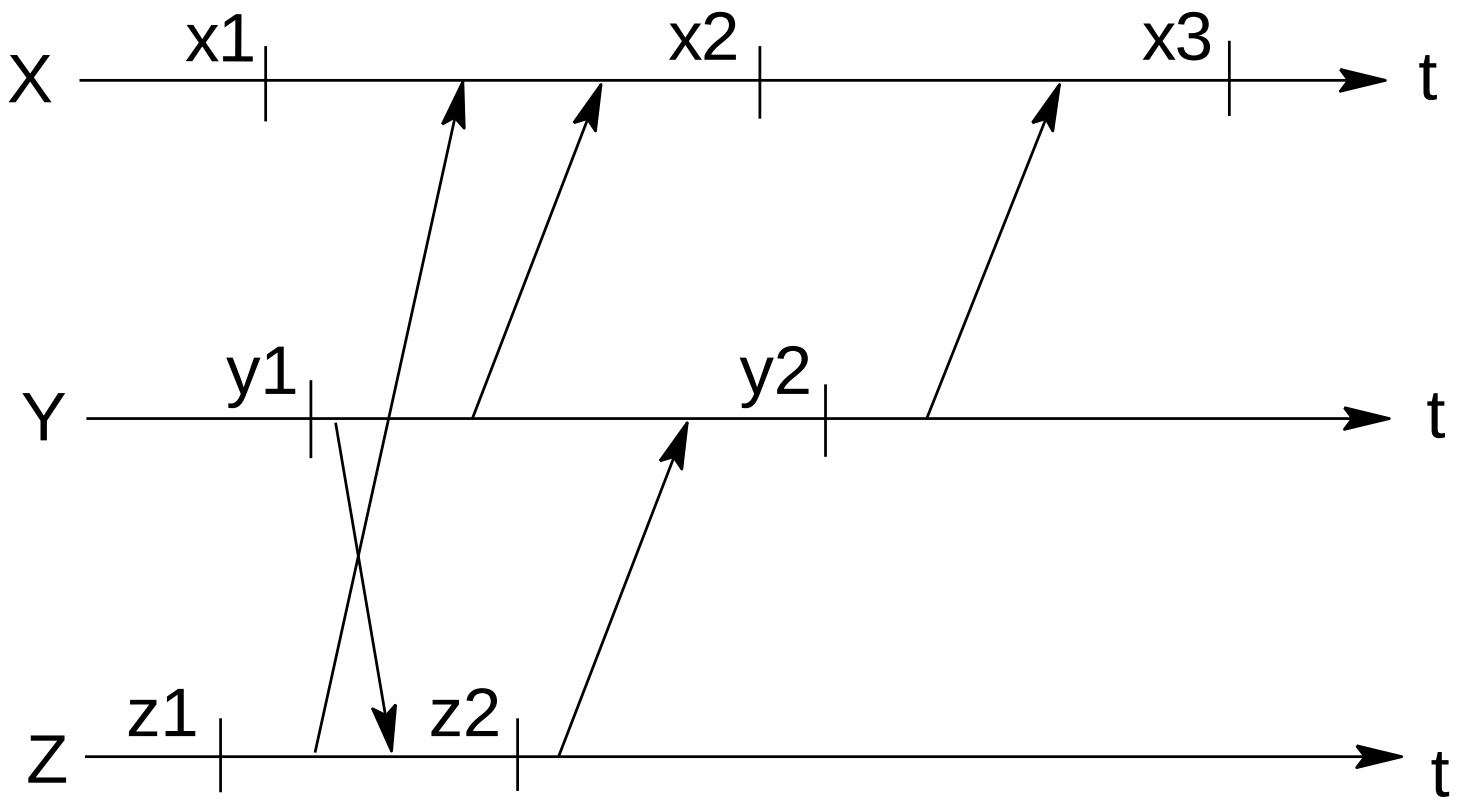
Возвратное восстановление

При возвратном восстановлении возникают следующие проблемы.

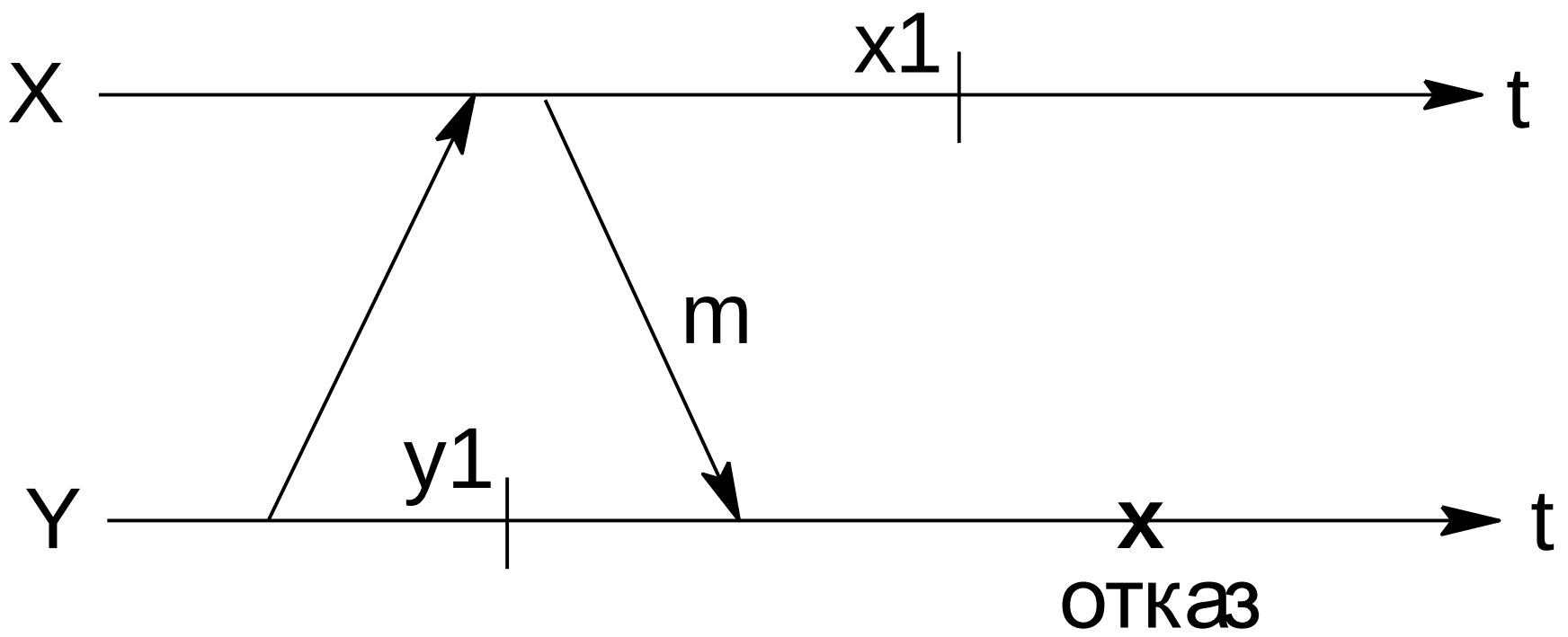
- Потери производительности, вызванные запоминанием состояний, восстановлением запомненного состояния и повторением ранее выполненной работы, могут быть слишком высоки.
- Нет гарантии, что сбой снова не повторится после восстановления.
- Для некоторых компонентов системы восстановление в предшествующее состояние может быть невозможно (торговый автомат).

  Для восстановления состояния в традиционных ЭВМ применяются два метода (и их комбинация), основанные на промежуточной фиксации состояния либо ведении журнала выполняемых операций. Они различаются объемом запоминаемой информацией и временем, требуемым для восстановления.

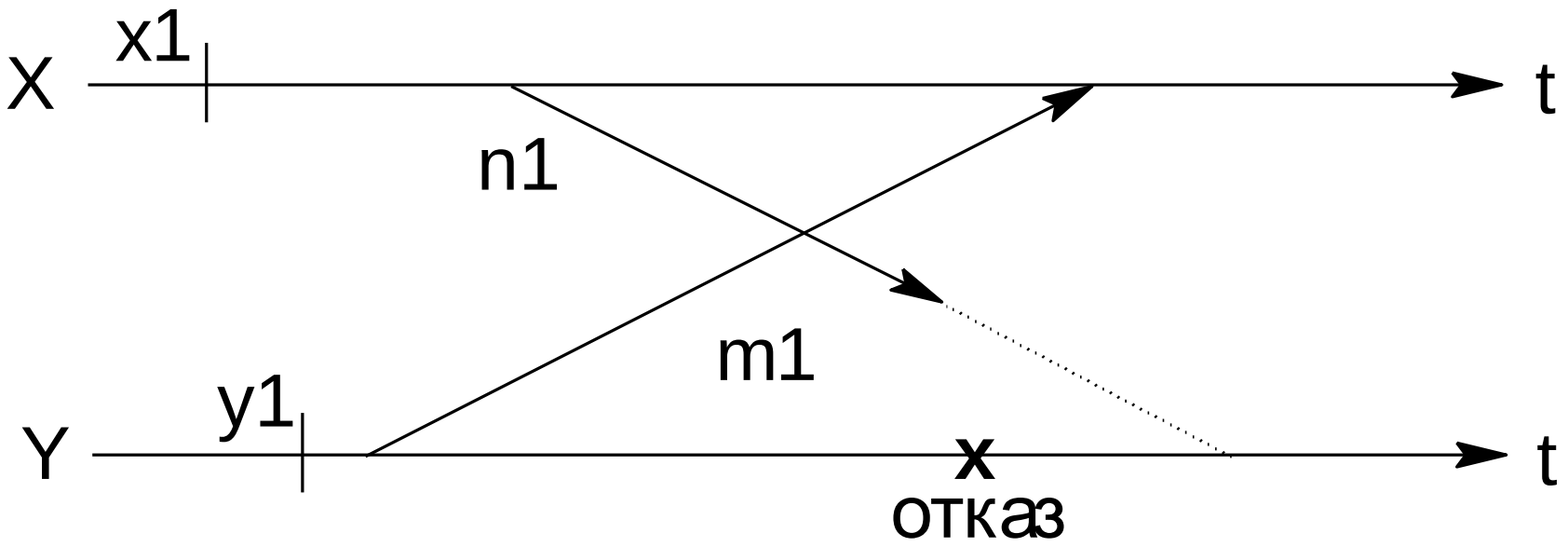
Сообщения сироты и эффект домино



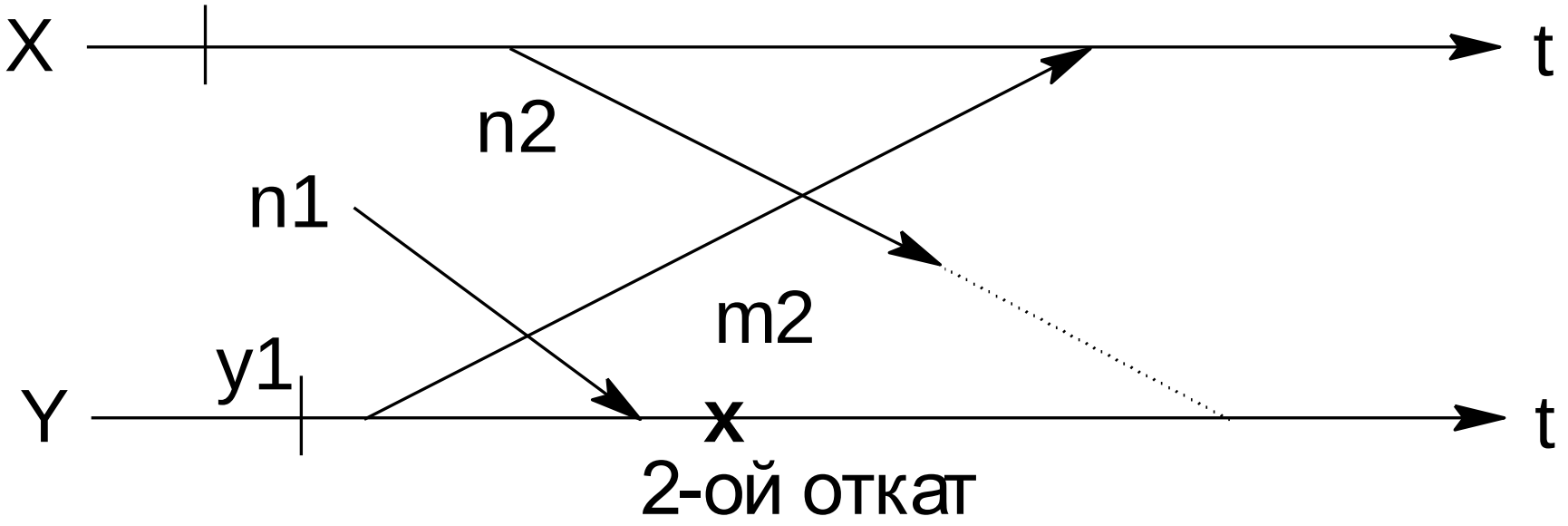
Потеря сообщений



Проблема бесконечного восстановления



Проблема бесконечного восстановления



n1 – сообщение-призрак

Консистентное множество контрольных точек

- Показанные ранее трудности показывают, что глобальная контрольная точка, состоящая из произвольной совокупности локальных контрольных точек, не обеспечивает восстановления взаимодействующих процессов.
- Для распределенных систем запоминание согласованного глобального состояния является серьезной теоретической проблемой.
- Множество контрольных точек называется **строго консистентным**, если во время его фиксации никаких обменов между процессами не было. Оно соответствует понятию строго консистентного глобального состояния, когда все посланные сообщения получены и нет никаких сообщений в каналах связи.
- Множество контрольных точек называется **консистентным**, если для любой зафиксированной операции приема сообщения, соответствующая операция посылки также зафиксирована (нет сообщений-сирот).

Простой метод фиксации консистентного множества контрольных точек

- Фиксация локальной контрольной точки после каждой операции посылки сообщения. При этом посылка сообщения и фиксация должны быть единой неделимой операцией (транзакцией). Множество последних локальных контрольных точек является консистентным (но не строго консистентным).
- Чтобы избежать потерь сообщений при восстановлении с использованием консистентного множества контрольных точек необходимо повторить отправку тех сообщений, квитанции о получении которых стали недействительными в результате отката. Используя временные метки сообщений можно распознавать сообщения-призраки и избежать бесконечного восстановления.

Синхронная фиксация контрольных точек и восстановление

К распределенной системе алгоритм предъявляет следующие требования.

1) Процессы взаимодействуют посредством посылки сообщений через коммуникационные каналы.
2) Каналы работают по алгоритму FIFO. Коммуникационные протоколы точка-точка гарантируют невозможность пропажи сообщений из-за ошибок коммуникаций или отката к контрольной точке.

Другой способ обеспечения этого - использование стабильной памяти для журнала посылаемых сообщений и фиксации идентификатора последнего полученного по каналу сообщения.

Синхронная фиксация контрольных точек и восстановление

1-ая фаза.

- Инициатор фиксации (процесс Pi) создает пробную контрольную точку и просит все остальные процессы сделать то же самое. При этом процессу запрещается посылать неслужебные сообщения после того, как он сделает пробную контрольную точку. Каждый процесс извещает Pi о том, сделал ли он пробную контрольную точку. Если все процессы сделали пробные контрольные точки, то Pi принимает решение о превращении пробных точек в постоянные. Если какой-либо процесс не смог сделать пробную точку, то принимается решение об отмене всех пробных точек.

Синхронная фиксация контрольных точек и восстановление

2-ая фаза.

- Pi информирует все процессы о своем решении. В результате либо все процессы будут иметь новые постоянные контрольные точки, либо ни один из процессов не создаст новой постоянной контрольной точки. Только после выполнения принятого процессом Pi решения все процессы могут посылать сообщения.
- Корректность алгоритма очевидна, поскольку созданное всеми множество постоянных контрольных точек не может содержать не зафиксированных операций посылки сообщений.
- Оптимизация: если процесс не посылал сообщения с момента фиксации предыдущей постоянной контрольной точки, то он может не создавать новую.

Синхронная фиксация контрольных точек и **восстановление**

Алгоритм восстановления предполагает, что его инициирует один процесс и он не будет выполняться параллельно с алгоритмом фиксации.

Выполняется в две фазы.

1) Инициатор отката спрашивает остальных, готовы ли они откатываться. Когда все будут готовы к откату, то он принимает решение об откате.
2) Pi сообщает всем о принятом решении. Получив это сообщение, каждый процесс поступает указанным образом. С момента ответа на опрос готовности и до получения принятого решения процессы не должны посылать сообщения (нельзя же посылать сообщение процессу, который уже мог успеть откатиться).

Асинхронная фиксация контрольных точек и восстановление

Синхронная фиксация упрощает восстановление, но связана с большими накладными расходами:

1) Дополнительные служебные сообщения для реализации алгоритма.
2) Синхронизационная задержка - нельзя посылать неслужебные сообщения во время работы алгоритма.

Если отказы редки, то указанные потери совсем не оправданы.

Фиксация может производиться асинхронно.

В этом случае множество контрольных точек может быть неконсистентным.

Асинхронная фиксация контрольных точек и восстановление

При откате происходит поиск подходящего консистентного множества путем поочередного отката каждого процесса в ту точку, в которой зафиксированы все посланные им и полученные другими сообщения (для ликвидации сообщений-сирот).

Алгоритм опирается на наличие в стабильной памяти для каждого процесса журнала, отслеживающего номера посланных и полученных им сообщений, а также на некоторые предположения об организации взаимодействия процессов, необходимые для исключения «эффекта домино» (например, организация приложения по схеме сообщение-реакция-ответ).

Отказоустойчивость

- Изложенные выше методы восстановления после отказов для некоторых систем непригодны (управляющие системы, транзакции в on-line режиме) из-за прерывания нормального функционирования.
- Чтобы избежать этих неприятностей, создают системы, устойчивые к отказам. Такие системы либо маскируют отказы, либо ведут себя в случае отказа заранее определенным образом (пример - изменения, вносимые транзакцией в базу данных, становятся невидимыми при отказе).
- Два механизма широко используются при обеспечении отказоустойчивости - *протоколы голосования* и *протоколы принятия коллективного решения.*

Протоколы принятия коллективного решения

Разделяются на два класса.

Протоколы принятия единого решения, в которых все исполнители являются исправными и должны либо все принять, либо все не принять заранее предусмотренное решение. Примерами такого решения являются решение о завершении итерационного цикла при достижении всеми необходимой точности, решение о реакции на отказ (этот протокол уже знаком нам - он использовался для принятия решения об откате всех процессов к контрольным точкам).

Протоколы принятия согласованных решений на основе полученных друг от друга данных. При этом необходимо всем исправным исполнителям получить достоверные данные от остальных исправных исполнителей, а данные от неисправных исполнителей проигнорировать.

Отказоустойчивость

Ключевой подход для обеспечения отказоустойчивости - избыточность (оборудования, процессов, данных).

- **Использование режима «горячего резерва»** (второй пилот, резервное ПО).

  Проблема переключения на резервный исполнитель.
- **Использование активного размножения**.

  Наглядный пример - тройное дублирование аппаратуры в бортовых компьютерах и голосование при принятии решения.

  Другие примеры – размножение страниц в DSM и размножение файлов в распределенных файловых системах.

Алгоритмы голосования

- Общая схема использования голосования при размножении файлов может быть представлена следующим образом.
- Файл может модифицироваться разными процессами только последовательно (при открытии файла на запись процесс-писатель будет ждать закрытия файла другим писателем или всеми читателями), а читаться всеми одновременно (протокол писателей-читателей). Все модификации файла нумеруются и каждая копия файла характеризуется номером версии – количеством ее модификаций.
- Каждой копии приписано некоторое количество голосов V_i.
- Пусть общее количество приписанных всем копиям голосов равно V
- Определяется кворум записи V_w и кворум чтения V_r так, что

  $V_w + V_r > V$

Алгоритмы голосования

- Для записи информации в файл писатель рассылает ее всем владельцам копий файла и должен получить V_w голосов от тех, кто успешно выполнил запись.
- Для получения права на чтение читателю достаточно получить необходимое число голосов (V_r) от любых серверов.
- Кворум чтения выбран так, что хотя бы один из тех серверов, от которых получено разрешение, является владельцем текущей копии файла. За чтением информации из файла читатель может обратиться к любому владельцу текущей копии файла.
- Описанная схема базируется на **статическом распределении голосов**. Различие в голосах, приписанных разным серверам, позволяет учесть их особенности (надежность, эффективность). Еще большую гибкость предоставляет метод **динамического перераспределения голосов**.
- Для того, чтобы выход из строя некоторых серверов не привел к ситуации, когда невозможно получить кворум, применяется механизм **изменения состава голосующих**.

Протоколы принятия единого решения

Рассмотрим известную «проблему двух армий».

- Армия зеленых численностью 5000 воинов располагается в долине.
- Две армии синих численностью по 3000 воинов находятся далеко друг от друга в горах, окружающих долину. Если две армии синих одновременно атакуют зеленых, то они победят. Если же в сражение вступит только одна армия синих, то она будет полностью разбита.
- Предположим, что командир 1-ой синей армии генерал Александр посылает (с посыльным) сообщение командиру 2-ой синей армии генералу Михаилу «Я имею план - давай атаковать завтра на рассвете». Посыльный возвращается к Александру с ответом Михаила - «Отличная идея, Саша. Увидимся завтра на рассвете». Александр приказывает воинам готовиться к атаке на рассвете.

Протоколы принятия единого решения

- Однако, чуть позже Александр вдруг осознает, что Михаил не знает о возвращении посыльного и поэтому может не отважиться на атаку. Тогда он отправляет посыльного к Михаилу чтобы подтвердить, что его (Михаила) сообщение получено Александром и атака должна состояться.
- Посыльный прибыл к Михаилу, но теперь тот боится, что не зная о прибытии посыльного Александр может не решиться на атаку. И т.д. Ясно, что генералы никогда не достигнут согласия.
- Предположим, что такой протокол согласия с конечным числом сообщений существует. Удалив избыточные последние сообщения, получим минимальный протокол. Самое последнее сообщение является существенным (поскольку протокол минимальный). Если это сообщение не дойдет по назначению, то войны не будет. Но тот, кто посылал это сообщение, не знает, дошло ли оно. Следовательно, он не может считать протокол завершенным и не может принять решение об атаке. Даже с надежными процессорами (генералами), принятие единого решения невозможно при ненадежных коммуникациях.

Задача «Византийских генералов»

- В этой задаче армия зеленых находится в долине, а ***n*** синих генералов возглавляют свои армии, расположенные в горах. Связь осуществляется по телефону и является надежной, но из ***n*** генералов ***m*** являются предателями. Предатели активно пытаются воспрепятствовать согласию лояльных генералов.
- Согласие в данном случае заключается в следующем. Каждый генерал знает, сколько воинов находится под его командой. Ставится цель, чтобы все лояльные генералы узнали численности всех лояльных армий, т.е. каждый из них получил один и тот же вектор длины ***n***, в котором i-ый элемент либо содержит численность i-ой армии (если ее командир лоялен) либо не определен (если командир предатель).
- Соответствующий рекурсивный алгоритм был предложен в 1982 г. (Lamport).
- Проиллюстрируем его для случая ***n***=4 и ***m***=1. В этом случае алгоритм осуществляется в 4 шага.

Задача «Византийских генералов»

- 1 шаг. Каждый генерал посылает всем остальным сообщение, в котором указывает численность своей армии. Лояльные генералы указывают истинное количество, а предатели могут указывать различные числа в разных сообщениях. Генерал-1 указал 1 (одна тысяча воинов), генерал-2 указал 2, генерал-3 указал трем остальным генералам соответственно x,y,z, а генерал-4 указал 4.
- 2-ой шаг. Каждый формирует свой вектор из имеющейся информации.

Получается:

vect1 (1,2,x,4)

vect2 (1,2,y,4)

vect3 (1,2,3,4)

vect4 (1,2,z,4)

- 3-ий шаг. Каждый посылает свой вектор всем остальным (генерал-3 посылает опять произвольные значения).

Задача «Византийских генералов»

- 3-ий шаг. Каждый посылает свой вектор всем остальным (генерал-3 посылает опять произвольные значения).

Генералы получают следующие вектора:

g1	g2	g3	g4
(1,2,y,4)	(1,2,x,4)	(1,2,x,4)	(1,2,x,4)
(a,b,c,d)	(e,f,g,h)	(1,2,y,4)	(1,2,y,4)
(1,2,z,4)	(1,2,z,4)	(1,2,z,4)	(i,j,k,l)

- 4-ый шаг. Каждый генерал проверяет каждый элемент во всех полученных векторах. Если какое-то значение совпадает по меньшей мере в двух векторах, то оно помещается в результирующий вектор, иначе соответствующий элемент результирующего вектора помечается «неизвестен».
- Все лояльные генералы получают один вектор (1,2,»неизвестен»,4) - согласие достигнуто.

Задача «Византийских генералов»

- Если рассмотреть случай n=3 и m=1, то согласие не будет достигнуто.
- Lamport доказал, что в системе с m неверно работающими процессорами можно достичь согласия только при наличии 2m+1 верно работающих процессоров (более 2/3).
- Другие авторы доказали, что в распределенной системе с асинхронными процессорами и неограниченными коммуникационными задержками согласие невозможно достичь даже при одном неработающем процессоре (даже если он не подает признаков жизни).
- Применение алгоритма - надежная синхронизация часов.

Алгоритм надежных неделимых широковещательных рассылок сообщений

- Алгоритм выполняется в две фазы и предполагает наличие в каждом процессоре очередей для запоминания поступающих сообщений. В качестве уникального идентификатора сообщения используется его начальный приоритет - логическое время отправления, значение которого на разных процессорах различно.

1-ая фаза

Процесс-отправитель посылает сообщение группе процессов (список их идентификаторов содержится в сообщении).

При получении этого сообщения процессы:

- Приписывают сообщению приоритет, помечают сообщение как «недоставленное» и буферизуют его. В качестве приоритета используется временная метка (текущее логическое время).
- Информируют отправителя о приписанном сообщению приоритете.

Алгоритм надежных неделимых широковещательных рассылок сообщений

2-ая фаза

При получении ответов от всех адресатов, отправитель:

- Выбирает из всех приписанных сообщению приоритетов максимальный и устанавливает его в качестве окончательного приоритета сообщения.
- Рассылает всем адресатам этот приоритет.

Получив окончательный приоритет, получатель:

- Приписывает сообщению этот приоритет.
- Помечает сообщение как «доставленное».
- Упорядочивает все буферизованные сообщения по возрастанию их приписанных приоритетов.
- Если первое сообщение в очереди отмечено как «доставленное», то оно будет обрабатываться как окончательно полученное.

Алгоритм надежных неделимых широковещательных рассылок сообщений

Если получатель обнаружит, что он имеет сообщение с пометкой «недоставленное», отправитель которого сломался, то он для завершения выполнения протокола осуществляет следующие два шага в качестве координатора.

1. Опрашивает всех получателей о статусе этого сообщения.

Получатель может ответить одним из трех способов:

- Сообщение отмечено как «недоставленное» и ему приписан такой-то приоритет.
- Сообщение отмечено как «доставленное» и имеет такой-то окончательный приоритет.
- Он не получал это сообщение.

Алгоритм надежных неделимых широковещательных рассылок сообщений

2. Получив все ответы координатор выполняет следующие действия:

- Если сообщение у какого-то получателя помечено как «доставленное», то его окончательный приоритет рассылается всем. (Получив это сообщение каждый процесс выполняет шаги фазы 2).
- Иначе координатор заново начинает весь протокол с фазы 1. (Повторная посылка сообщения с одинаковым приоритетом не должна вызывать коллизий).

Необходимо заметить, что алгоритм требует хранения начального и окончательного приоритетов даже для принятых и уже обработанных сообщений.

https://drive.google.com/drive/folders/192NoY0y8bdVu-ukka3kEGcxuC3Zo1IYH?usp=sharing

Суперкомпьютеры петафлопсной производительности

#12 Top500 (June'19, Rmax = 17.590 PFlop/s); 10/12 — 08/19
200 cabinets; 18,688 nodes; 299,008 cores; 18,688 K20X; 693.6TBytes;
8.2 MW;

R. A. Ashraf and C. Engelmann, **Analyzing the Impact of System Reliability Events on Applications in the Titan Supercomputer**, 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS), 2018

Event log from Reliability, Availability and Serviceability (RAS) System

$68k$ apps have 1+ events (over $2 \cdot 10^6$)

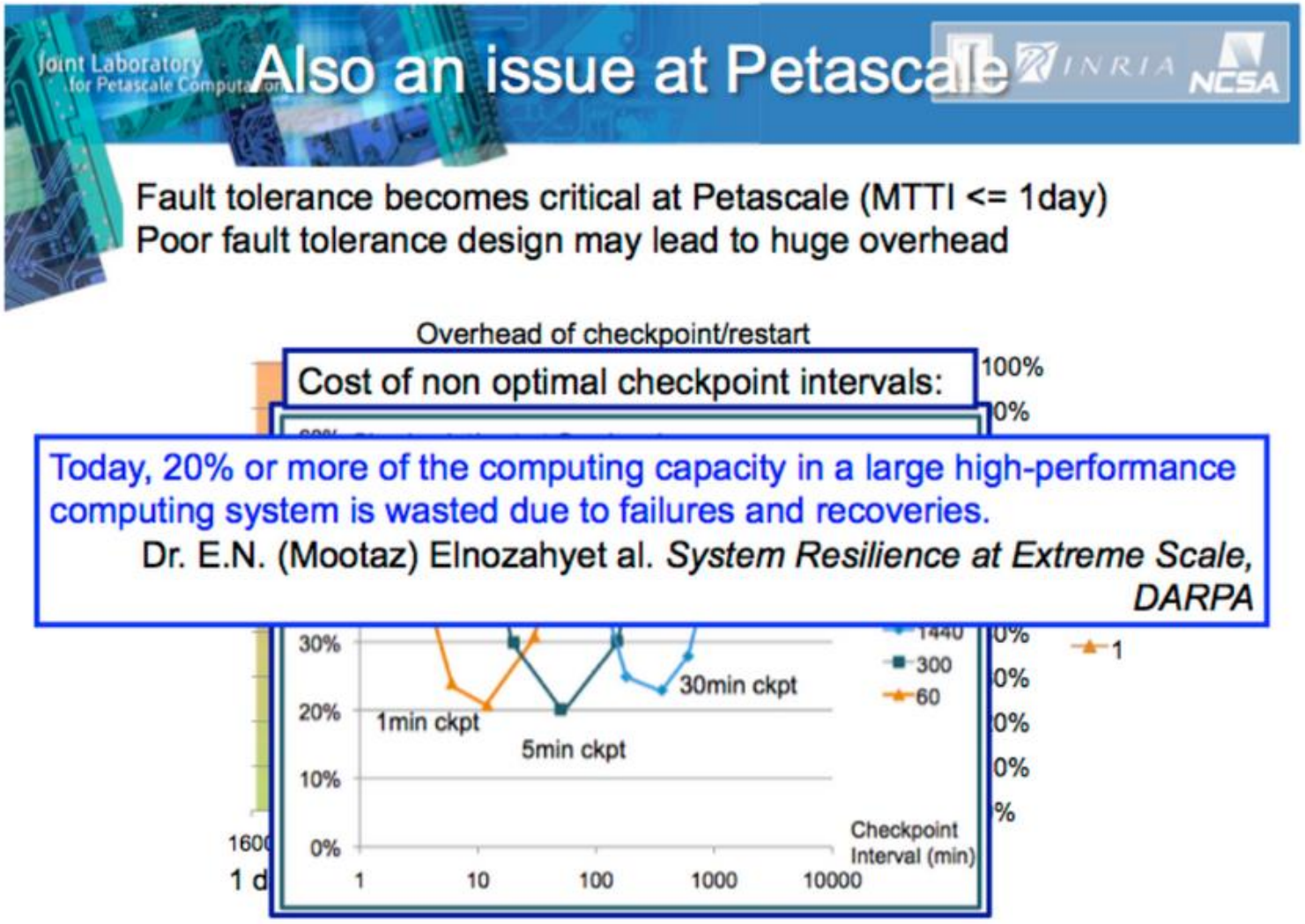
95% of apps with 11k+ nodes have 1+ events
91% of apps with 125- nodes have 0 events
85% apps under 30min have 0 events
80% apps above 24h have 1+ events

Nature of first event hitting an application:

Parallel Filesystem	73.7%
Processor Failure	15.7 %
Machine Check Exception	6.5%
GPU Failure	1.5%
OOM / SEGFAULT	1.9%
Interconnect	0.8 %

http://fault-tolerance.org/sc21/

Суперкомпьютеры петафлопсной производительности



http://fault-tolerance.org/sc21/

Суперкомпьютеры экзафлопсной производительности

- Pre-exascale machines are hierarchical, many with accelerators
  - Top 10: $2 \cdot 10^4$ to $1.5 \cdot 10^5$ nodes
  - Each node equipped with 48 to 256 cores
  - Many of those featuring multiple (up to 6) GPU or other accelerator / node
- Exascale machines
  - Many will feature accelerators
  - Number of nodes should remain in the same order of magnitude
  - Level of parallelism needs to reach 1 billion threads at 1GHz each
- Failure-prone

MTBF – one component	1 year	10 years	120 years
MTBF – platform of 10^4 components	1 min	9 min	4 days
MTBF – platform of 10^5 components	6 sec	1 min	10h
MTBF – platform of 10^6 components	$<$ 1 sec	6 sec	1h

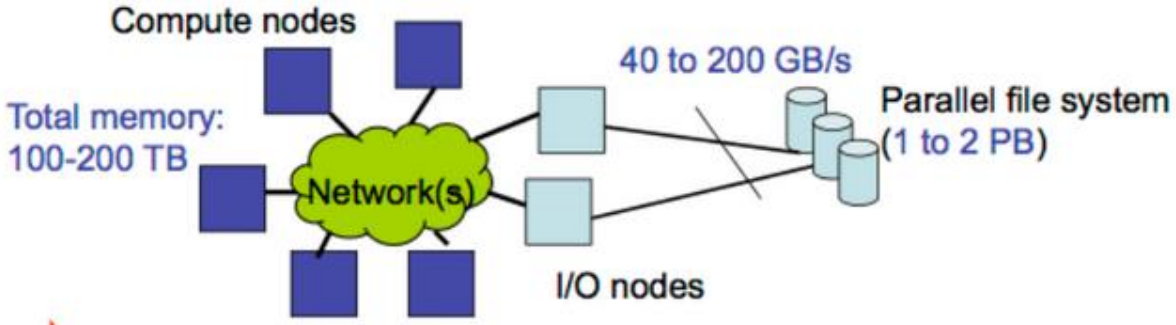
More hardware component ⇒ Shorter MTBF (Mean Time Between Failures)

http://fault-tolerance.org/sc21/

Контрольные точки

Classic approach for FT: Checkpoint-Restart

Typical "Balanced Architecture" for PetaScale Computers



RoadRunner

Without optimization, Checkpoint-Restart needs about 1h! (~30 minutes each)

Systems	Perf.	Ckpt time	Source
RoadRunner	1PF	~20 min.	Panasas
LLNL BG/L	500 TF	>20 min.	LLNL
LLNL Zeus	11TF	26 min.	LLNL
YYY BG/P	100 TF	~30 min.	YYY

LLNL BG/L

http://fault-tolerance.org/sc21/

Обработка ошибок в OpenMP-программах

Директива

#pragma omp cancel *clause[[,] clause]*

где clause одна из:

- **parallel**
- **sections**
- **for**
- **taskgroup**
- **if (***scalar-expression***)**

Директива

#pragma omp cancellation point *clause[[,] clause]*

где clause одна из:

- **parallel**
- **sections**
- **for**
- **taskgroup**

Новая функция системы поддержки:

- **omp_get_cancellation**

Новая переменная окружения:

- **OMP_CANCELLATION**

Обработка ошибок в OpenMP-программах

```
void example() {
  std::exception *ex = NULL;
  #pragma omp parallel shared(ex)
  {
    #pragma omp for schedule (dynamic)
    for (int i = 0; i < N; i++) {
      try {
        causes_an_exception();
      } catch (std::exception *e) {
        #pragma omp atomic write
          ex = e; // still must remember exception for later handling
        #pragma omp cancel for // cancel worksharing construct
      }
    }
    if (ex) { // if an exception has been raised, cancel parallel region
      #pragma omp cancel parallel
    }
  }
  if (ex) {  // handle exception stored in ex
  }
}
```

Поиск в дереве

```
typedef struct binary_tree_s {
  int value;
  struct binary_tree_s *left, *right;
} binary_tree_t;

binary_tree_t *search_tree_parallel (binary_tree_t *tree, int value) {
   binary_tree_t *found = NULL;
#pragma omp parallel shared(found, tree, value)
   {
#pragma omp taskgroup
      {
#pragma omp master
         {
            found = search_tree(tree, value, 0);
         }
      }
   }
   return found;
}
```

Поиск в дереве

```
binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
   binary_tree_t *found = NULL;
   if (tree) {
      if (tree->value == value) {
         found = tree;
      } else {
         #pragma omp task shared(found) if(level < 10)
         {
            binary_tree_t *found_left = NULL;
            found_left = search_tree(tree->left, value, level + 1);
            if (found_left) {
               #pragma omp atomic write
                 found = found_left;
               #pragma omp cancel taskgroup
            }
         }
```

Поиск в дереве

```
#pragma omp task shared(found) if(level < 10)
        {
           binary_tree_t *found_right = NULL;
           found_right = search_tree(tree->right, value, level + 1);
           if (found_right) {
              #pragma omp atomic write
                 found = found_right;
              #pragma omp cancel taskgroup
           }
        }
        #pragma omp taskwait
     }
  }
  return found;
}
```

Сбой во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
{
   int rank, size, rc, len;
   char errstr[MPI_MAX_ERROR_STRING];
   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Barrier(MPI_COMM_WORLD);
   if( rank == (size-1) ) raise(SIGKILL);
   rc = MPI_Barrier(MPI_COMM_WORLD);
   MPI_Error_string(rc, errstr, &len);
   printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n", rank, size, errstr);
   MPI_Finalize();
}
```

Сбой во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
{
    int rank, size, rc, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    rc = MPI_Barrier(MPI_COMM_WORLD);
    MPI_Error_string(rc, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n", rank, size, errstr);
    MPI_Finalize();
}
```

Сбой во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void verbose_errhandler(MPI_Comm* comm, int* err, ...) {
    int rank, size, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Error_string( *err, errstr, &len );
    printf("Rank %d / %d: Notified of error %s\n",
           rank, size, errstr);
}
```

Сбой во время работы MPI-программы

```
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Errhandler errh;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_create_errhandler(verbose_errhandler, &errh);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Rank %d / %d: Stayin' alive!\n", rank, size);
    MPI_Finalize();
}
```

User Level Failure Mitigation (ULFM)

https://fault-tolerance.org/

#include <mpi-ext.h>

- **MPIX_ERR_PROC_FAILED** when a process failure prevents the completion of an MPI operation.
- **MPIX_ERR_PROC_FAILED_PENDING** when a potential sender matching a non-blocking wildcard source receive has failed.
- **MPIX_ERR_REVOKED** when one of the ranks in the application has invoked the MPI_Comm_revoke operation on the communicator.
- **MPIX_Comm_revoke(MPI_Comm comm)** Interrupts any communication pending on the communicator at all ranks.
- **MPIX_Comm_shrink(MPI_Comm comm, MPI_Comm* newcomm)** creates a new communicator where dead processes in comm were removed.
- **MPIX_Comm_agree(MPI_Comm comm, int *flag)** performs a consensus (i.e. fault tolerant allreduce operation) on flag (with the operation bitwise or).
- **MPIX_Comm_failure_get_acked(MPI_Comm, MPI_Group*)** obtains the group of currently acknowledged failed processes.
- **MPIX_Comm_failure_ack(MPI_Comm)** acknowledges that the application intends to ignore the effect of currently known failures on wildcard receive completions and agreement return values.

Сбой во время работы MPI-программы

```
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Errhandler errh;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_create_errhandler(verbose_errhandler, &errh);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}
```

Сбой во время работы MPI-программы

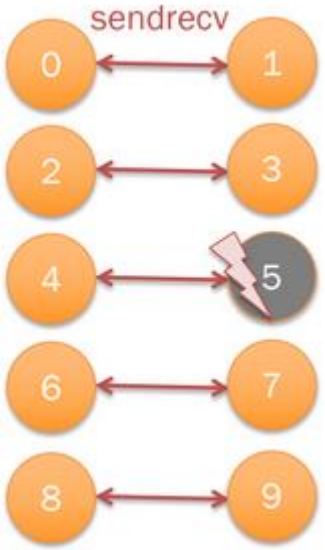
```
static void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...) {
    MPI_Comm comm = *pcomm;
    int err = *perr;
    char errstr[MPI_MAX_ERROR_STRING];
    int i, rank, size, nf, len, eclass;
    MPI_Group group_c, group_f;
    int *ranks_gc, *ranks_gf;

    MPI_Error_class(err, &eclass);
    if( MPIX_ERR_PROC_FAILED != eclass ) {
        MPI_Abort(comm, err);
    }
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
```

Сбой во время работы MPI-программы

```
    /* We use a combination of 'ack/get_acked' to obtain the list of failed processes.     */
    MPIX_Comm_failure_ack(comm);
    MPIX_Comm_failure_get_acked(comm, &group_f);
    MPI_Group_size(group_f, &nf);
    MPI_Error_string(err, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. %d found dead: { ", rank, size, errstr, nf);
    /* We use 'translate_ranks' to obtain the ranks of failed procs  in 'comm‘ communicator  */
    ranks_gf = (int*)malloc(nf * sizeof(int));
    ranks_gc = (int*)malloc(nf * sizeof(int));
    MPI_Comm_group(comm, &group_c);
    for(i = 0; i < nf; i++)
        ranks_gf[i] = i;
    MPI_Group_translate_ranks(group_f, nf, ranks_gf,
                              group_c, ranks_gc);
    for(i = 0; i < nf; i++)
        printf("%d ", ranks_gc[i]);
    printf("}\n");
    free(ranks_gf); free(ranks_gc);
}
```

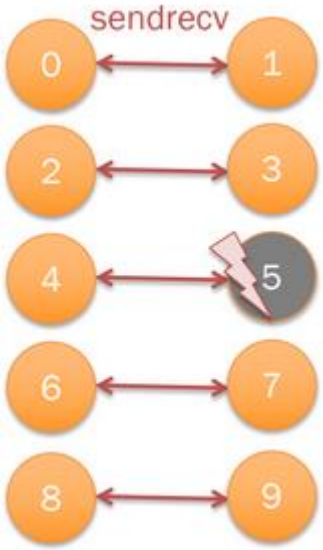
Сбой во время работы MPI-программы

```
int main(int argc, char *argv[]) {
    int rank, size, peer;
    MPI_Errhandler errh;
    double myvalue, hisvalue=NAN;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_create_errhandler(verbose_errhandler, &errh);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD,  errh);
    MPI_Barrier(MPI_COMM_WORLD);
    myvalue = rank/(double)size;
    if( 0 == rank%2 )
        peer = ((rank+1)<size)? rank+1: MPI_PROC_NULL;
    else
        peer = rank-1;
```



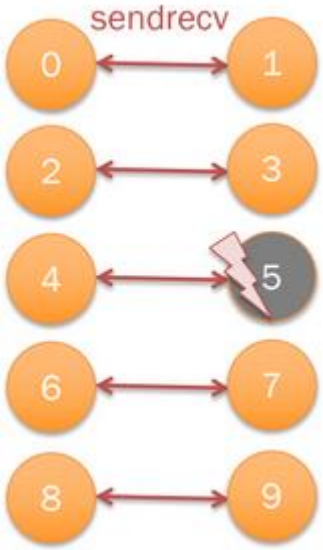
Сбой во время работы MPI-программы

```
    if( rank == (size/2) ) raise(SIGKILL);
   /* exchange a value between a pair of two consecutive
    * odd and even ranks; not communicating with anybody
    * else. */
   MPI_Sendrecv(&myvalue, 1, MPI_DOUBLE, peer, 1,
                &hisvalue, 1, MPI_DOUBLE, peer, 1,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
   if( peer != MPI_PROC_NULL)
      printf("Rank %d / %d: value from %d is %g\n",
             rank, size, peer, hisvalue);
   MPI_Finalize();
}
```



Сбой во время работы MPI-программы

```
    if( rank == (size/2) ) raise(SIGKILL);
    /* exchange a value between a pair of two consecutive
     * odd and even ranks; not communicating with anybody
     * else. */
    MPI_Sendrecv(&myvalue, 1, MPI_DOUBLE, peer, 1,
                 &hisvalue, 1, MPI_DOUBLE, peer, 1,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    if( peer != MPI_PROC_NULL)
       printf("Rank %d / %d: value from %d is %g\n",
              rank, size, peer, hisvalue);
    MPI_Finalize();
}
```




```
bash$ $ULFM_PREFIX/bin/mpirun  -np 10 03.undisturbed
Rank 0 / 10: value from 1 is 0.1
Rank 1 / 10: value from 0 is 0
Rank 3 / 10: value from 2 is 0.2
Rank 2 / 10: value from 3 is 0.3
Rank 6 / 10: value from 7 is 0.7
Rank 7 / 10: value from 6 is 0.6
Rank 9 / 10: value from 8 is 0.8
Rank 8 / 10: value from 9 is 0.9
Rank 4 / 10: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead: { 5 }
Rank 4 / 10: value from 5 is nan
```

Sendrecv between pairs of live processes complete w/o error. Can post more, it will work too!

Sendrecv failed at rank 4 (5 is dead)
Value not updated!

Сбой во время работы MPI-программы

```
void error_handler(MPI_Comm *pcomm, int *error_code, ...) {
    int rc, i, myrank, oldrank, num_fails, error_class;
   char error_string[MPI_MAX_ERROR_STRING];
   MPI_Group f_group, w_group;
   int f_group_rank[MAX_SIZE], w_group_rank[MAX_SIZE];
   MPI_Comm communicator = *pcomm, new_comm;
    …
   /* I found a failed process, so I will acknowledge the failure and shrink the
      communicator */
    if(0 == myrank) {
        //    Get group from current communicator
        MPI_Comm_group(communicator, &w_group);
        //    Get group of failed procs
        MPIX_Comm_failure_ack(communicator);
        MPIX_Comm_failure_get_acked(communicator, &f_group);
```

Сбой во время работы MPI-программы

```
    // Get no. of failed procs
    MPI_Group_size(f_group, &num_fails);
    // Get ranks of failed procs in the original comm
    for(i = 0; i < num_fails; i++) f_group_rank[i] = i;
    MPI_Group_translate_ranks(f_group, num_fails, f_group_rank,
                              w_group, w_group_rank);
    // Stop the workers. Prevent a communicator from being used in the future
    MPIX_Comm_revoke(communicator);
  }
  //   Creates a new communitor from an existing communicator while excluding
  //   failed processes
  MPIX_Comm_shrink(communicator, &new_comm);
  communicator = new_comm;
  MPI_Comm_size(communicator, &myrank);
  ...
}
```

https://drive.google.com/file/d/1A2bC4qCHtBLbWAQQS_7IcWyINAsUrsrb/view

Обработка ошибок в MPI-программах

```
static void compute(int how_long);
int main(int argc, char* argv[]) {
    /* Send some large data to make the recv long enough to see something
      interesting */
    int count = 4*1024*1024;
    int* send_buff = malloc(count * sizeof(*send_buff));
    int* recv_buff = malloc(count * sizeof(*recv_buff));
    /* Every process sends a token to the right on a ring (size tokens sent per
     * iteration, one per process per iteration) */
    int left = MPI_PROC_NULL, right = MPI_PROC_NULL;
    int rank = MPI_PROC_NULL, size = 0;
    int tag = 42, iterations = 10, how_long = 5;
    MPI_Request req = MPI_REQUEST_NULL;
    double post_date, wait_date, complete_date;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    left = (rank+size-1)%size; /* left ring neighbor in circular space */
    right = (rank+size+1)%size; /* right ring neighbor in circular space */
```

Обработка ошибок в MPI-программах

```
/**** Done with initialization, main loop now ***********************/
printf("Entering test; computing silently for %d seconds\n", how_long);
for(int i = 0; i < iterations; i++) {
  MPI_Barrier(MPI_COMM_WORLD);
  post_date = MPI_Wtime();
  MPI_Irecv(recv_buff, 512, MPI_INT, left, tag, MPI_COMM_WORLD, &req);
  MPI_Send (send_buff, 512, MPI_INT, right, tag, MPI_COMM_WORLD);
  compute(how_long);
  wait_date = MPI_Wtime();
  MPI_Wait(&req, MPI_STATUS_IGNORE);
  complete_date = MPI_Wtime();
  printf("Worked %g seconds before entering MPI_Wait and spend %g seconds
  waiting\n",   wait_date - post_date,  complete_date - wait_date);
}
MPI_Finalize();
return 0;
}
```

Обработка ошибок в MPI-программах

```
/**** Done with initialization, main loop now ***********************/
printf("Entering test; computing silently for %d seconds\n", how_long);
for(int i = 0; i < iterations; i++) {
  MPI_Barrier(MPI_COMM_WORLD);
  post_date = MPI_Wtime();
  MPI_Irecv(recv_buff, 512, MPI_INT, left, tag, MPI_COMM_WORLD, &req);
  MPI_Send (send_buff, 512, MPI_INT, right, tag, MPI_COMM_WORLD);
  compute(how_long);
  wait_date = MPI_Wtime();
  MPI_Wait(&req, MPI_STATUS_IGNORE);
  complete_date = MPI_Wtime();
  printf("Worked %g seconds before entering MPI_Wait and spend %g seconds
  waiting\n",   wait_date - post_date,  complete_date - wait_date);
}
MPI_Finalize();
return 0;
}
```

```
mpirun -mca mpi_ft_detector_timeout 1 -np 10 test
Entering test; computing silently for 5 seconds ...
[saturn:62859] *** An error occurred in MPI_Send *** reported by process  1
[saturn:62859] *** on communicator MPI_COMM_WORLD
[saturn:62859] *** MPI_ERR_PROC_FAILED: Process Failure
[saturn:62859] *** MPI_ERRORS_ARE_FATAL (processes in this
communicator will now abort, and potentially your MPI job)
```

Использование дополнительных процессов

MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)

IN	comm	-	родительский коммуникатор;
IN	color	-	признак подгруппы;
IN	key	-	управление упорядочиванием;
OUT	newcomm	-	новый коммуникатор.

Функция расщепляет группу, связанную с родительским коммуникатором, на непересекающиеся подгруппы по одной на каждое значение признака подгруппы color.
Значение color должно быть неотрицательным. Каждая подгруппа содержит процессы с одним и тем же значением color.
Параметр key управляет упорядочиванием внутри новых групп: меньшему значению key соответствует меньшее значение идентификатора процесса.
В случае равенства параметра key для нескольких процессов упорядочивание выполняется в соответствии с порядком в родительской группе.

Использование дополнительных процессов

```
MPI_comm comm, newcomm;
int myid, color;
. . . .
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split(comm, color, myid, &newcomm);
```