

MARIUSZ GIL

MODELING
COMPLEX
PROCESSES
AND TIME WITH

SAGA PATTERN

[Source Ministry]



BO·TT·EGA
IT minds



you.

MODELING
COMPLEX
PROCESSES
AND TIME WITH

SAGA PATTERN

MODELING
COMPLEX
PROCESSES
AND TIME WITH

SAGA PATTERN

MODELING
COMPLEX
PROCESSES
AND TIME WITH

SAGA PATTERN

MODELING
COMPLEX
PROCESSES
AND TIME WITH

SAGA PATTERN

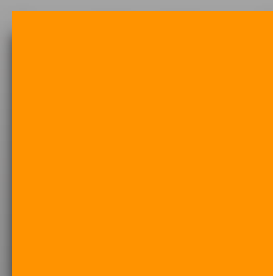
MODELING
COMPLEX
PROCESSES
AND TIME WITH

SAGA PATTERN

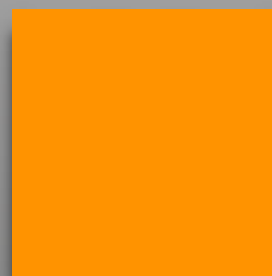
MODELING

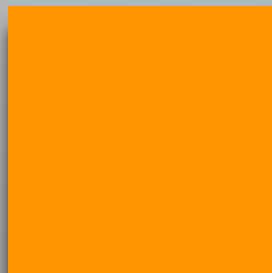
EVENT MODELING

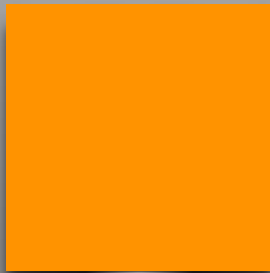
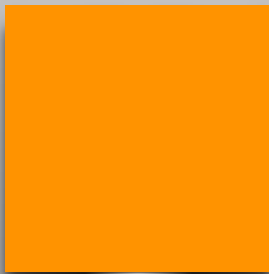
EVENT STORMING

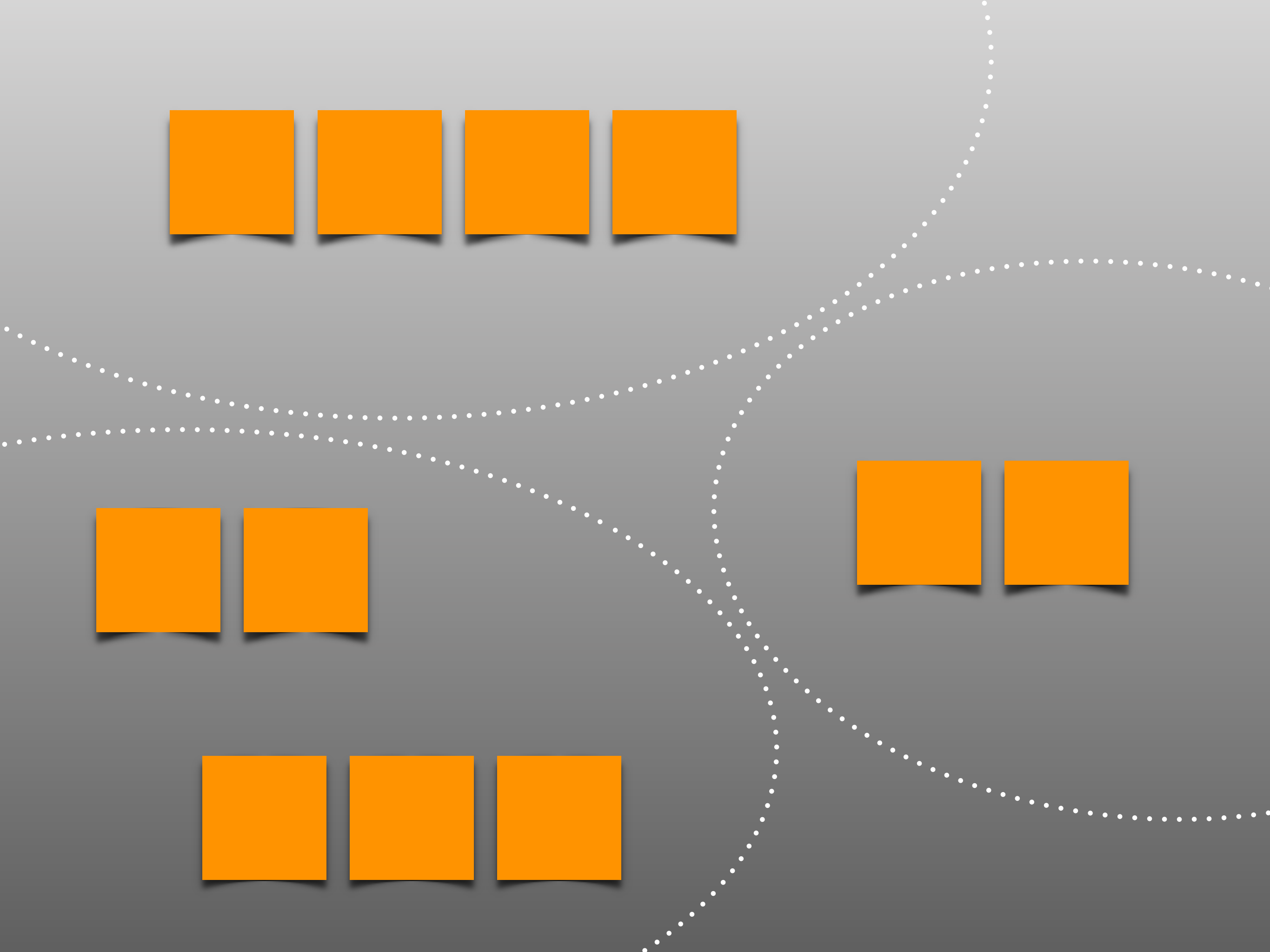


EVENT

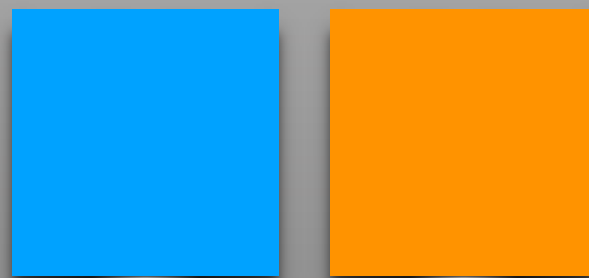




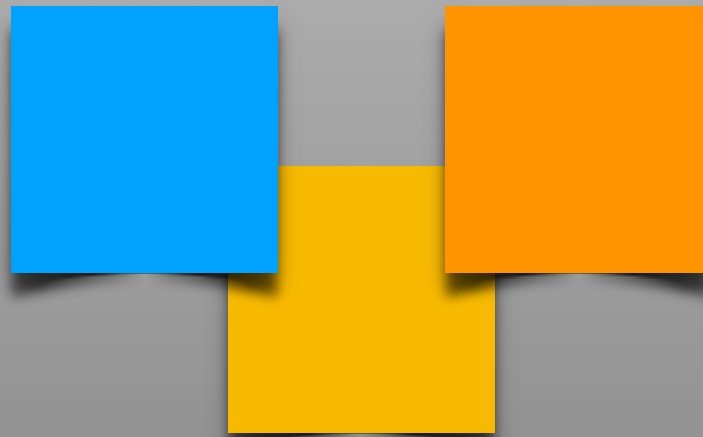




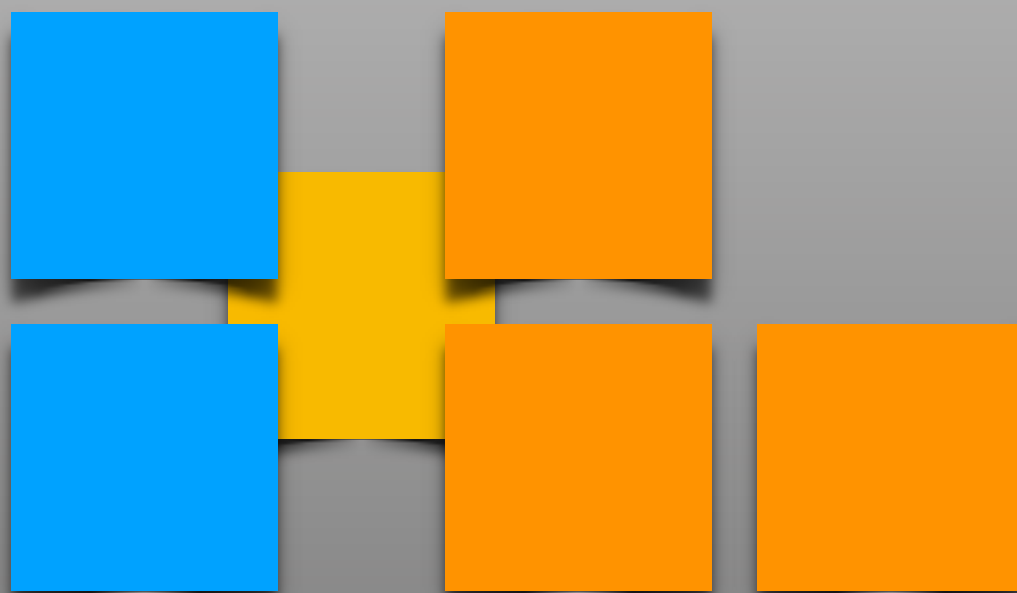


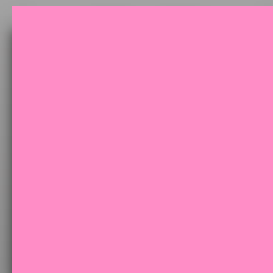


COMMAND / EVENT



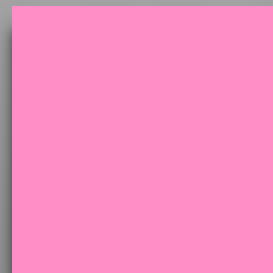
COMMAND / AGGREGATE / EVENT





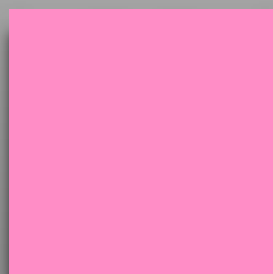
POLICY

/ REACTIVE LOGIC /

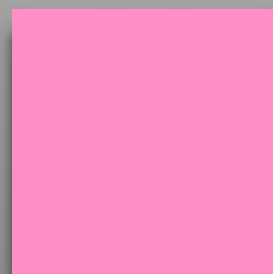


POLICY

/ LISTENER / SAGA / PROCESS MANAGER /



LISTENER



SAGA



SAGAS

Hector Garcia-Molina
Kenneth Salem

Department of Computer Science
Princeton University
Princeton, N J 08544

Abstract

Long lived transactions (LLTs) hold on to database resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions. To alleviate these problems we propose the notion of a saga. A LLT is a saga if it can be written as a sequence of transactions that can be interleaved with other transactions. The database management system guarantees that either all the transactions in a saga are successfully completed or compensating transactions are run to amend a partial execution. Both the concept of saga and its implementation are relatively simple, but they have the potential to improve performance significantly. We analyze the various implementation issues related to sagas, including how they can be run on an existing system that does not directly support them. We also discuss techniques for database and LLT design that make it feasible to break up LLTs into sagas.

1. INTRODUCTION

As its name indicates, a *long lived transaction* is a transaction whose execution, even without interference from other transactions, takes a substantial amount of time, possibly on the order of hours or days. A long lived transaction, or LLT, has a long duration compared to

the majority of other transactions either because it accesses many database objects, it has lengthy computations, it pauses for inputs from the users, or a combination of these factors. Examples of LLTs are transactions to produce monthly account statements at a bank, transactions to process claims at an insurance company, and transactions to collect statistics over an entire database [Gray81a].

In most cases, LLTs present serious performance problems. Since they are transactions, the system must execute them as atomic actions, thus preserving the consistency of the database [Date81a, Ullm82a]. To make a transaction atomic, the system usually locks the objects accessed by the transaction until it commits, and this typically occurs at the end of the transaction. As a consequence, other transactions wishing to access the LLT's objects suffer a long locking delay. If LLTs are long because they access many database objects then other transactions are likely to suffer from an increased blocking rate as well, i.e. they are more likely to conflict with an LLT than with a shorter transaction.

Furthermore, the transaction abort rate can also be increased by LLTs. As discussed in [Gray81b], the frequency of deadlock is very sensitive to the "size" of transactions, that is, to how many objects transactions access. (In the analysis of [Gray81b] the deadlock frequency grows with the fourth power of the transaction size.) Hence, since LLTs access many objects, they may cause many deadlocks, and correspondingly, many abortions. From the point of view of system crashes, LLTs have a higher probability of encountering a failure (because of their duration), and are thus more likely to encounter yet more delays and more likely to be aborted themselves.

the saga tables to discover the status of pending sagas. This scan would be performed by submitting a database transaction. The TEC will only execute this transaction after transaction recovery is complete, hence the SD will read consistent data. Once the SD knows the status of the pending sagas, it issues the necessary compensating or normal transactions, just as the SEC would have after recovery. Care must be taken not to interfere with sagas that started right after the crash, but before the SD submitted its database query.

After the TEC aborts a transaction (e.g., because of a deadlock or a user initiated abort), it may simply kill the process that initiated the transaction. In a conventional system this may be fine, but with sagas this leaves the saga unfinished. If the TEC cannot signal the SD when this occurs, then the SD will have to periodically scan the saga table searching for such a situation. If found, the corrective action is immediately taken.

A running saga can also directly request services from the SD. For instance, to perform an abort-saga, the abort-saga subroutine sends the request to the SD and then (if necessary) executes an abort-transaction.

8. PARALLEL SAGAS

Our model for sequential transaction execution within a saga can be extended to include parallel transactions. This could be useful in an application where the transactions of a saga are naturally executed concurrently. For example, when processing a purchase order, it may be best to generate the shipping order and update accounts receivable at the same time.

We will assume that a saga process (the parent) can create new processes (children) with which it will run in parallel, with a request similar to a fork request in UNIX. The system may also provide a join capability to combine processes within a saga.

Backward crash recovery for parallel sagas is similar to that for sequential sagas. Within each process of the parallel saga, transactions are compensated for (or undone) in reverse order just as with sequential sagas. In addition, all compensations in a child process must occur before any compensations for transactions in the parent that were executed before the child was created (forked). (Note that only transaction execution order *within a process* and fork and join informa-

tion constrain the order of compensation. If T_1 and T_2 have executed in parallel processes and T_2 has read data written by T_1 , compensating for T_1 does not force us to compensate for T_2 first.)

Unlike backward crash recovery, backward recovery from a saga failure is more complicated with parallel sagas because the saga may consist of several processes, all of which must be terminated. For this, it is convenient to route all process fork and join operations through the SEC so it can keep track of the process structure of the saga. When one of the saga processes requests an abort-saga, the SEC kills all processes involved in the saga. It then aborts all pending transactions and compensates all committed ones.

Forward recovery is even more complicated due to the possibility of "inconsistent" save-points. To illustrate, consider the saga of Figure 8.1. Each box represents a process, within each box is the sequence of transactions and save-points (sp) executed by the process. The lower process was forked after T_1 committed. Suppose that T_3 and T_5 are the currently executing transactions and that save-points were executed before T_1 and T_5 .

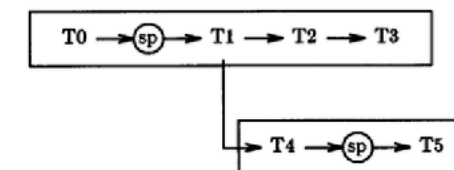


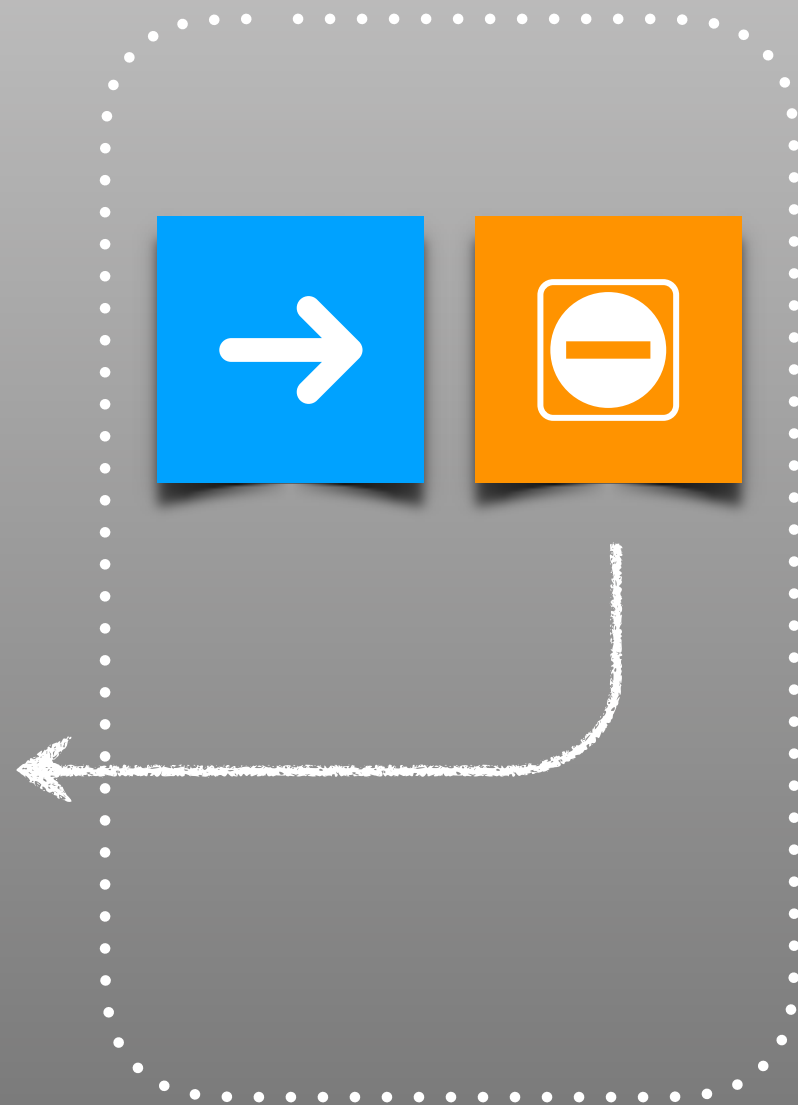
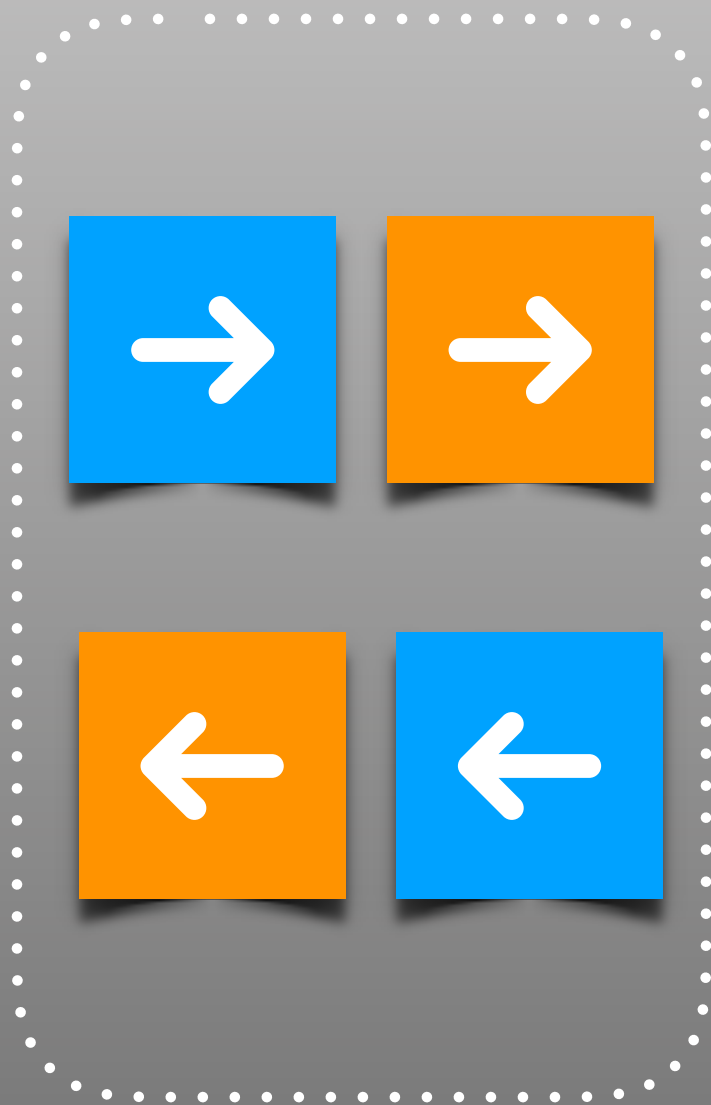
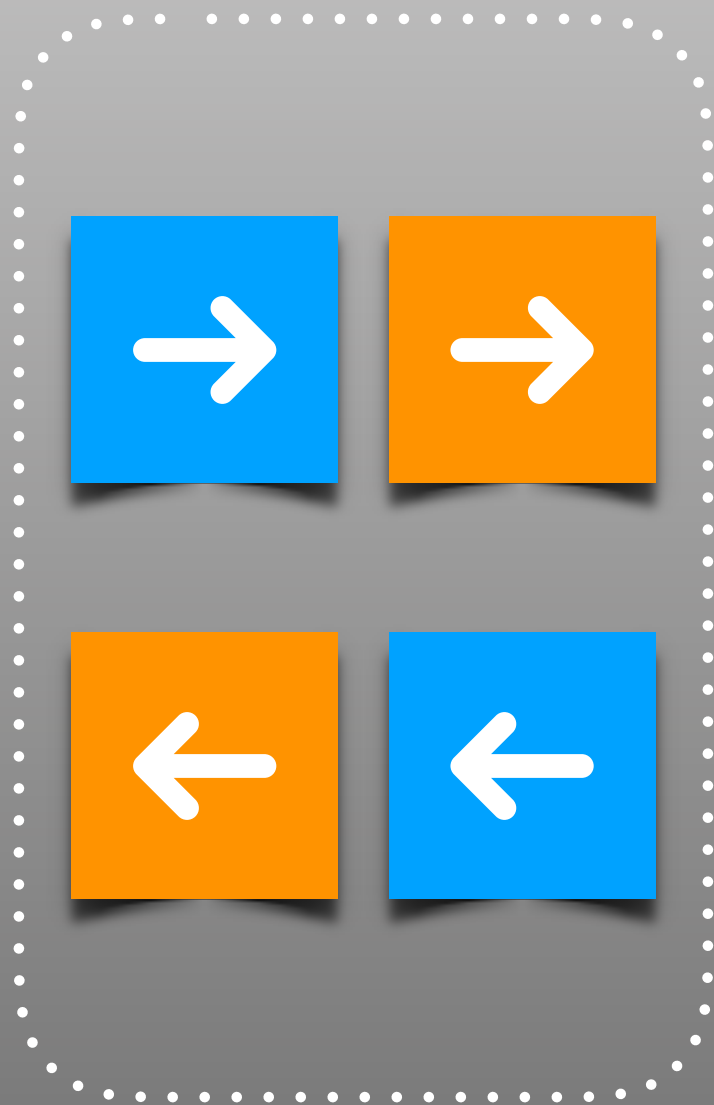
Figure 8.1 - Parallel Saga

At this point the system fails. The top process will have to be restarted before T_1 . Therefore, the save-point made by the second process is not useful. It depends on the execution of T_1 which is being compensated for.

This problem is known as cascading roll backs. It has been analyzed in a scenario where processes communicate via messages or shared data objects [Hadz82a, Rand78a]. There it is possible to analyze save-point dependencies to arrive at a consistent set of save-points (if it exists). The consistent set can then be used to restart the processes. With parallel sagas, the situation is even simpler since save-point dependencies arise only through forks and joins, and

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

A Saga is a distribution of multiple workflows across multiple systems, each providing a path (fork) of compensating actions in the event that any of the steps in the workflow fails



ACID

/ ATOMICITY / CONSISTENCY / ISOLATION / DURABILITY /

BASE

/ BASIC AVAILABILITY / SOFT STATE / EVENTUAL CONSISTENCY /

FAILURE PATTERN

BROADWAY




```
/**
 * Event
 */
class OrderPlaced
{
    private $orderId;
    private $numberOfSeats;

    public function __construct($orderId, $numberOfSeats)
    {
        $this->orderId = $orderId;
        $this->numberOfSeats = $numberOfSeats;
    }

    public function orderId()
    {
        return $this->orderId;
    }

    public function numberOfSeats()
    {
        return $this->numberOfSeats;
    }
}
```

```
/**
 * Command
 */
class MakeSeatReservation
{
    private $reservationId;
    private $numberOfSeats;

    public function __construct($reservationId, $numberOfSeats)
    {
        $this->reservationId = $reservationId;
        $this->numberOfSeats = $numberOfSeats;
    }

    public function reservationId()
    {
        return $this->reservationId;
    }

    public function numberOfSeats()
    {
        return $this->numberOfSeats;
    }
}
```



```
/**
 * Event
 */
class ReservationAccepted
{
    private $reservationId;

    public function __construct($reservationId)
    {
        $this->reservationId = $reservationId;
    }

    public function reservationId()
    {
        return $this->reservationId;
    }
}
```

```
/**
 * Event
 */
class ReservationRejected
{
    private $reservationId;

    public function __construct($reservationId)
    {
        $this->reservationId = $reservationId;
    }

    public function reservationId()
    {
        return $this->reservationId;
    }
}
```

```
/**
 * Command
 */
class MarkOrderAsBooked
{
    private $orderId;

    public function __construct($orderId)
    {
        $this->orderId = $orderId;
    }
}
```

```
/**
 * Command
 */
class RejectOrder
{
    private $orderId;

    public function __construct($orderId)
    {
        $this->orderId = $orderId;
    }
}
```

```
class ReservationSaga extends Saga
    implements StaticallyConfiguredSagaInterface
{
    private $commandBus;
    private $uuidGenerator;

    public function __construct(
        CommandBus $commandBus,
        UuidGeneratorInterface $uuidGenerator
    ) {
        $this->commandBus = $commandBus;
        $this->uuidGenerator = $uuidGenerator;
    }

    // ...
}
```

```
class ReservationSaga extends Saga
    implements StaticallyConfiguredSagaInterface
{
    // ...

    public static function configuration()
    {
        return [
            'OrderPlaced' => function (OrderPlaced $event) {
                return null; // no criteria, start of a new saga
            },
            'ReservationAccepted' => function (ReservationAccepted $event) {
                // return a Criteria object to fetch the State of this saga
                return new Criteria([
                    'reservationId' => $event->reservationId()
                ]);
            },
            'ReservationRejected' => function (ReservationRejected $event) {
                // return a Criteria object to fetch the State of this saga
                return new Criteria([
                    'reservationId' => $event->reservationId()
                ]);
            }
        ];
    }
}
```

```
class ReservationSaga extends Saga
    implements StaticallyConfiguredSagaInterface
{
    // ...

    public function handleOrderPlaced(
        OrderPlaced $event,
        State $state
    ) {
        // keep the order id,
        // for reference in `handleReservationAccepted()`
        // and `handleReservationRejected()`
        $state->set('orderId', $event->orderId());

        // generate an id for the reservation
        $reservationId = $this->uuidGenerator->generate();
        $state->set('reservationId', $reservationId);

        // make the reservation
        $this->commandBus->dispatch(
            new MakeSeatReservation($reservationId, $event->numberOfSeats())
        );

        return $state;
    }
}
```

```
class ReservationSaga extends Saga
    implements StaticallyConfiguredSagaInterface
{
    // ...

    public function handleReservationAccepted(
        ReservationAccepted $event,
        State $state
    ) {
        // the seat reservation for the given order is has been accepted,
        // mark the order as booked
        $this->commandBus->dispatch(
            new MarkOrderAsBooked($state->get('orderId'))
        );

        // the saga ends here
        $state->setDone();

        return $state;
    }
}
```



```
class ReservationSaga extends Saga
    implements StaticallyConfiguredSagaInterface
{
    // ...

    public function handleReservationRejected(
        ReservationRejected $event,
        State $state
    ) {
        // the seat reservation for the given order is has been rejected,
        // reject the order as well
        $this->commandBus->dispatch(
            new RejectOrder($state->get('orderId'))
        );

        // the saga ends here
        $state->setDone();

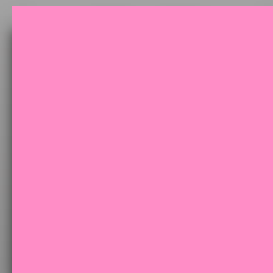
        return $state;
    }
}
```

```
class ReservationSagaTest extends SagaScenarioTestCase
{
    protected function createSaga(CommandBus $commandBus)
    {
        return new ReservationSaga(
            $commandBus,
            new MockUuidSequenceGenerator(
                [
                    'bf142ea0-29f7-11e5'
                ]
            ));
    }

    /**
     * @test
     */
    public function it_makes_a_seat_reservation_when_an_order_was_placed()
    {
        $this->scenario
            ->when(new OrderPlaced('9d66f760-29f7-11e5', 5))
            ->then([
                new MakeSeatReservation('bf142ea0-29f7-11e5', 5)
            ]);
    }
}
```

```
class ReservationSagaTest extends SagaScenarioTestCase
{
    // ...

    /**
     * @test
     */
    public function it_rejects_order_when_seat_reservation_was_rejected()
    {
        $this->scenario
            ->given([
                new OrderPlaced('9d66f760-29f7-11e5', 5)
            ])
            ->when(new ReservationRejected('bf142ea0-29f7-11e5'))
            ->then([
                new RejectOrder('9d66f760-29f7-11e5')
            ]);
    }
}
```



PROCESS MANAGER

/ ROUTING SLIP /

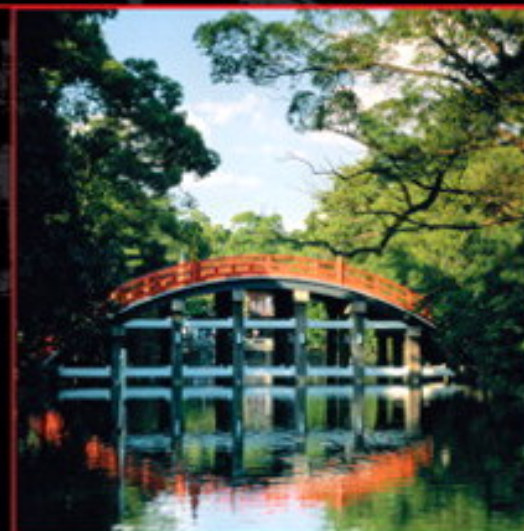
The Addison-Wesley Signature Series

ENTERPRISE INTEGRATION PATTERNS

DESIGNING, BUILDING, AND
DEPLOYING MESSAGING SOLUTIONS

GREGOR HOHPE
BOBBY WOOLF

WITH CONTRIBUTIONS BY
KYLE BROWN
CONRAD F. D'CRUZ
MARTIN FOWLER
SEAN NEVILLE
MICHAEL J. RETTIG
JONATHAN SIMON

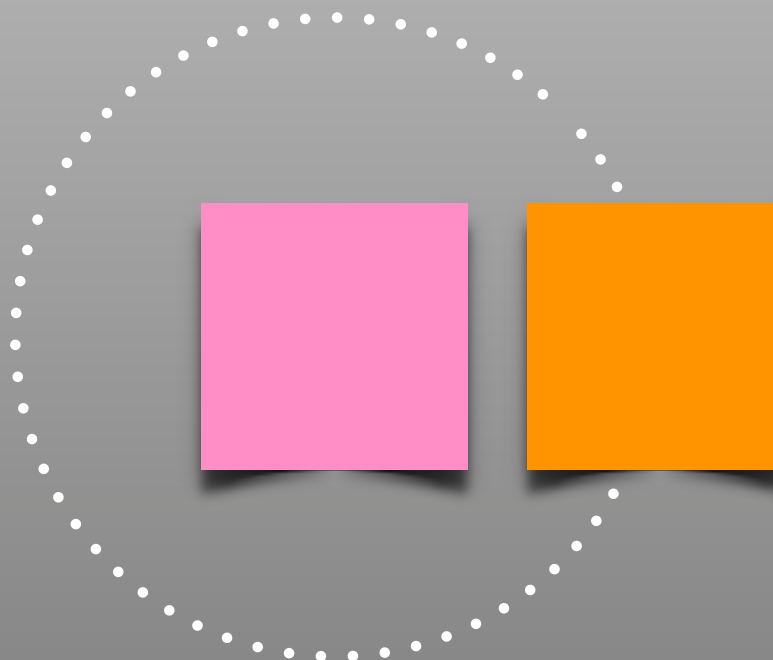
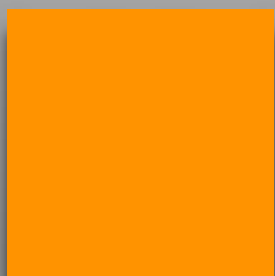
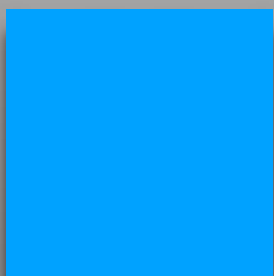
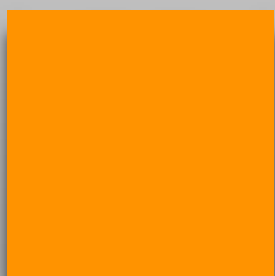
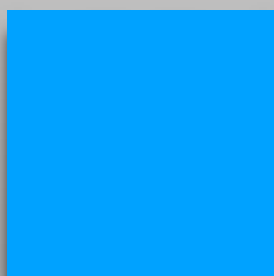


Forewords by John Crupi and Martin Fowler



A MARTIN FOWLER SIGNATURE
BOOK
Martin

A Process Manager is a central processing unit that maintains the state of the sequence of processing steps and determines the next step based on intermediate results



The background is a complex, abstract pattern in shades of blue and teal. It features swirling, organic shapes that resemble tentacles or the skin of an octopus. The pattern is dense and textured, with many small, dark, teardrop-shaped elements scattered throughout. The overall effect is a vibrant, underwater-themed background.

OCTOPUS PATTERN

TAKEAWAYS

THANKS!

ANY QUESTIONS?

 [mariuszgil](#)