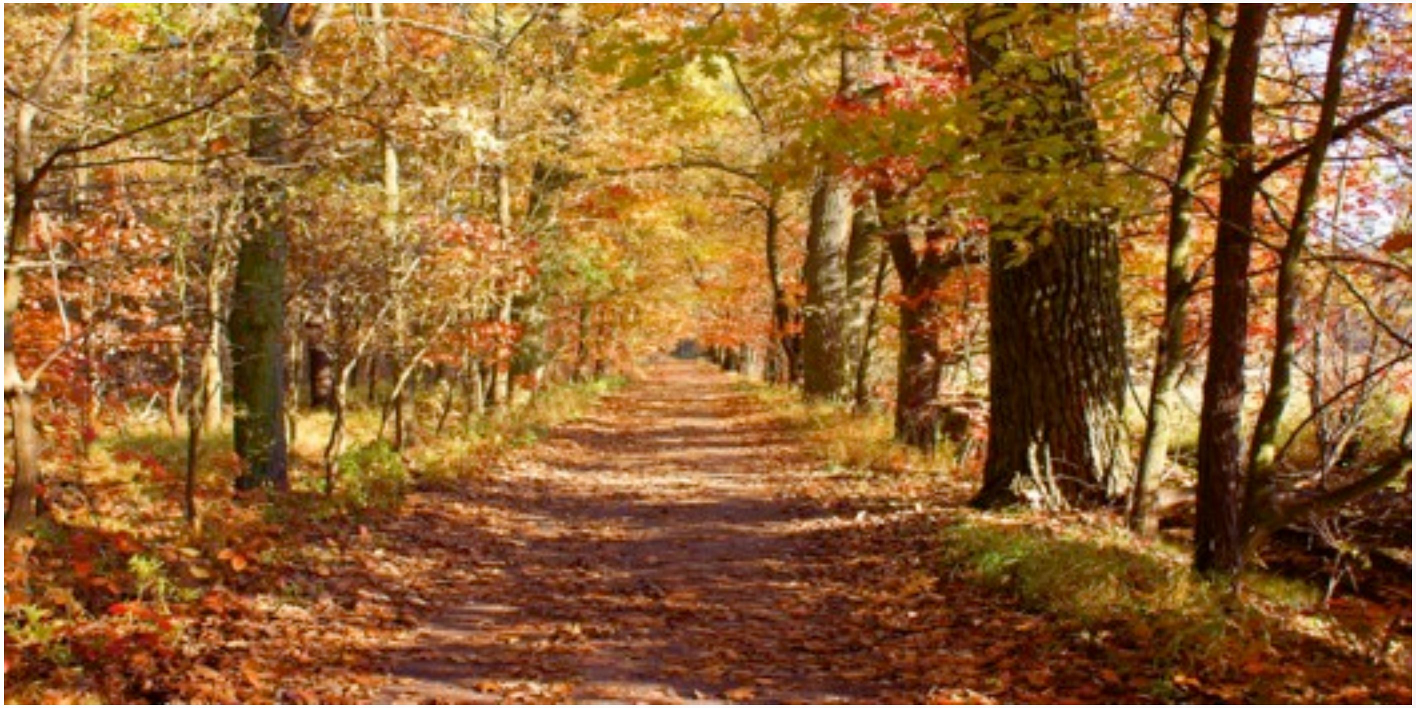
# The Draft Specification of Transactional Language Constructs




## BoostCon 2011

**Justin Gottschlich**
**Programming Systems Lab**
**Intel Labs**



## Acknowledgements

**Hans Boehm, Victor Luchangco, Mark Moir, Tatiana Shpeisman, Michael Wong**

# Motivation

# Motivation

- Bridge Boost and C++ TM Spec
  - Tutorial of C++ TM Spec
  - Solicit feedback, volunteer scribe?

# Outline

# Outline

- TM and the C++ TM Spec
  - Violations, TM, and examples
  - Achievements and future work

# Outline

- TM and the C++ TM Spec
  - Violations, TM, and examples
  - Achievements and future work
- Audience discussion throughout

# Concurrency

# Concurrency

- Data race

# Concurrency

- Data race
- Atomicity violation

# Concurrency

- Data race
- Atomicity violation
- Order violation

# Concurrency

- Data race
- Atomicity violation
- Order violation
- Deadlock

# Concurrency

- Data race
- Atomicity violation
- Order violation
- Deadlock
- Livelock

# TM in a Nutshell

# TM in a Nutshell

- What is TM?
  - **Open-ended** concurrency control paradigm
  - **Transactions**
  - Guarantees **atomicity** and **isolation**
  - **Composable**

# TM is Open-Ended

# TM is Open-Ended

- Optimistic (speculative), pessimistic

# TM is Open-Ended

- Optimistic (speculative), pessimistic
- Non-blocking, lock-based

# TM is Open-Ended

- Optimistic (speculative), pessimistic
- Non-blocking, lock-based
- Parallel, distributed

# TM Uses Transactions

# TM Uses Transactions

- Finite sequence of operations

# TM Uses Transactions

- Finite sequence of operations
- Begin, commit, cancel / abort

# TM Uses Transactions

- Finite sequence of operations
- Begin, commit, cancel / abort

```
// x == 0, y == 1
void swap(int &x, int &y)
{
    __transaction
    {
        int tmp = x;
        x = y;
        y = tmp;
    }
} // x == 1, y == 0
```

# Atomicity and

# Atomicity and

- Atomic
  - All or nothing

# Atomicity and

- Atomic
  - All or nothing
- Isolated
  - State prior to commit is invisible

# Atomicity and

- Atomic
  - All or nothing
- Isolated
  - State prior to commit is invisible

```
// x == 0, y == 1
void swap(int &x, int &y)
{
    __transaction
    {
        int tmp = x;
        x = y;
        y = tmp;
    }
} // x == 1, y == 0;
```

# Atomicity and

- Atomic
  - All or nothing

**x = 1, y = 1 invisible wrt transactions** →

- Isolated
  - State prior to commit is invisible

```
// x == 0, y == 1
void swap(int &x, int &y)
{
    __transaction
    {
        int tmp = x;
        x = y;
        y = tmp;
    }
} // x == 1, y == 0;
```

# Atomicity and

- Atomic
  - All or nothing

**x = 1, y = 1**
**invisible**
**wrt transactions**

- Isolated
  - State prior to commit is invisible

```
// x == 0, y == 1
void swap(int &x, int &y)
{
    __transaction
    {
        int tmp = x;
        x = y;
        y = tmp;
    }
} // x == 1, y == 0;
```

**atomic**

# Composition

# Composition

```
class Account
{
public:
    void withdraw(int amt)
    {
        lock(l_);
        bal_ -= amt;
        unlock(l_);
    }
    void deposit(int amt)
    {
        lock(l_);
        bal_ += amt;
        unlock(l_);
    }
private:
    int bal_; lock l_;
};
```

# Composition

```
class Account
{
public:
    void withdraw(int amt)
    {
        lock(l_);
        bal_ -= amt;
        unlock(l_);
    }
    void deposit(int amt)
    {
        lock(l_);
        bal_ += amt;
        unlock(l_);
    }
private:
    int bal_; lock l_;
};
```

```
int Account::balance()
{
    lock(l_);
    int val = bal_;
    unlock(l_);
    return val;
}
```

# Composition

```
class Account
{
public:
   void withdraw(int amt)
   {
      lock(l_);
      bal_ -= amt;
      unlock(l_);
   }
   void deposit(int amt)
   {
      lock(l_);
      bal_ += amt;
      unlock(l_);
   }
private:
   int bal_; lock l_;
};
```

```
int Account::balance()
{
   lock(l_);
   int val = bal_;
   unlock(l_);
   return val;
}
```

```
Account chk, sav;

void transfer(int amt)
{
   chk.withdraw(amt);
   sav.deposit(amt);
}
void bal(int &c, int &s)
{
   c = chk.balance();
   s = sav.balance();
}
```

# Composition

```
cla
{
pub



pri

};
```

```
// Initial state: chk.bal_ = 100;
//                sav.bal_ =   0;
//
// Move $100 from checking to savings
//
// chk.bal_ = 100, sav.bal_ =   0
// chk.bal_ =   0, sav.bal_ = 100


Thread T1                  Thread T2
------------------         ------------------
chk.withdraw(100);
                           chk.balance(); // 0
                           sav.balance(); // 0
sav.deposit(100);
```

time

```
s)
    c = chk.balance();
    s = sav.balance();
}
```

# Composition

```
// Initial state: chk.bal_ = 100;
//                sav.bal_ =   0;
//
// Move $100 from checking to savings
//
// chk.bal_ = 100, sav.bal_ =   0
// chk.bal_ =   0, sav.bal_ = 100


Thread T1                  Thread T2
------------------         ------------------
chk.withdraw(100);
                           chk.balance(); // 0
                           sav.balance(); // 0
sav.deposit(100);
```

time

```
cla
{
pub

pri

};
```

```
s)
    c = chk.balance();
    s = sav.balance();
}
```



Wednesday, May 18, 2011

# Composition: Take

# Composition: Take

- Transactions can be combined

# Composition: Take

- Transactions can be combined
- Many small txes → one big tx

# Composition: Take

- Transactions can be combined
- Many small txes → one big tx
- Big tx remains atomic / isolated

# Composition: Take

```
class Account
{
public:
  void withdraw(int amt){
    __transaction
    { bal_ -= amt; }
  }
  void deposit(int amt){
    __transaction
    { bal_ += amt; }
  }
  int balance(){
    __transaction
    { return bal_; }
  }
private:
  int bal_;
};
```

- Transactions can be combined
- Many small txes → one big tx
- Big tx remains atomic / isolated

# Composition: Take

```
class Account
{
public:
  void withdraw(int amt){
    __transaction
    { bal_ -= amt; }
  }
  void deposit(int amt){
p
  int bal_;
};
```

```
void bal(int &c, int &s)
{
   __transaction
   {
     c = chk.balance();
     s = sav.balance();
   }
}
```

- Transactions can be combined
- Many small txes → one big tx
- Big tx remains atomic / isolated

```
Account chk, sav;

void transfer(int amt)
{
   __transaction
   {
     chk.withdraw(amt);
     sav.deposit(amt);
   }
}
```

# Composition: Take

```
class Account
{
pub
  v
  
  
  }
  
vo
{



}
p
  i
};
```

- Transactions can be combined
  ...e big tx

```
// Move $100 from checking to savings
//
// chk.bal_ = 100, sav.bal_ =   0
// chk.bal_ =   0, sav.bal_ = 100

Thread T1                  Thread T2
------------------         ------------------
                           chk.balance(); // 100
                           sav.balance(); // 0
chk.withdraw(100);
sav.deposit(100);
                           chk.balance(); // 0
                           sav.balance(); // 100
```

# Composition: Take

```
class Account
{
pub
   v
   }
```

- Transactions can be combined ... e big tx

```
// Move $100 from checking to savings
//
// chk.bal_ = 100, sav.bal_ =   0
// chk.bal_ =   0, sav.bal_ = 100

Thread T1                  Thread T2
------------------         ------------------
                           chk.balance(); // 100
                           sav.balance(); // 0
chk.withdraw(100);
sav.deposit(100);
                           chk.balance(); // 0
                           sav.balance(); // 100
```

```
};
```

# Making Locks

# Making Locks

```
class Account
{
public:
    void withdraw(int amt);
    void deposit(int amt);
    void lock();
    void unlock();
};

void transfer(int amt)
{
    chk.lock();
    sav.lock();
    chk.withdraw(amt);
    sav.deposit(amt);
    ... // do unlock
}
```

# Making Locks

```
class Account
{
public:
    void withdraw(int amt);
    void deposit(int amt);
    void lock();
    void unlock();
};

void transfer(int amt)
{
    chk.lock();
    sav.lock();
    chk.withdraw(amt);
    sav.deposit(amt);
    ... // do unlock
}
```

- Problems
  - Exposes implementation
  - Deadlock?
  - Degrades to coarse-grained locking

# Making Locks

```
class Account
{
public:
   void withdraw(int amt);
   void deposit(int amt);
   void lock();
   void unlock();
};

void transfer(int amt)
{
   chk.lock();
   sav.lock();
   chk.withdraw(amt);
   sav.deposit(amt);
   ... // do unlock
}
```

- Problems
  - Exposes implementation
  - Deadlock?
  - Degrades to coarse-grained locking

# Making Locks

```
class Account
{
public:
    void withdraw(int amt);
    void deposit(int amt);
    void lock();
    void unlock();
};

void transfer(int amt)
{
    chk.lock();
    sav.lock();
    chk.withdraw(amt);
    sav.deposit(amt);
    ... // do unlock
}
```

- Problems
  - Exposes implementation
  - Deadlock?
  - Degrades to coarse-grained locking

# Making Locks

```
class Account
{
public:
    void withdraw(int amt);
    void deposit(int amt);
    void lock();
    void unlock();
};

void transfer(int amt)
{
    chk.lock();
    sav.lock();
    chk.withdraw(amt);
    sav.deposit(amt);
    ... // do unlock
}
```

- Problems
  - Exposes implementation
  - Deadlock?
  - Degrades to coarse-grained locking

```
void transfer(int amt)
{
    sav.lock();
    chk.lock();
    sav.deposit(amt);
    chk.withdraw(amt);
    ... // do unlock
}
```

# Making Locks

```
class Account
{
public:
   void withdraw(int amt);
   void deposit(int amt);
   void lock();
   void unlock();
};

void transfer(int amt)
{
   chk.lock();
   sav.lock();
   chk.withdraw(amt);
   sav.deposit(amt);
   ... // do unlock
}
```

- Problems
  - Exposes implementation
  - Deadlock?
  - Degrades to coarse-grained locking

```
void transfer(int amt)
{
   sav.lock();
   chk.lock();
   sav.deposit(amt);
   chk.withdraw(amt);
   ... // do unlock
}
```

# Examples of the C++ TM

# An Interesting

# An Interesting

```
class Id
{
public:
   Id(size_t id) : id_(id) {}
private:
   size_t const id_;
};

class Account : public Id
{
public:
   Account() : Id(count++) {}
private:
   static size_t count = 0;
};
```

# An Interesting

```
class Id
{
public:
    Id(size_
private:
    size_t const id_;
};

class Account : public Id
{
public:
    Account() : Id(count++) {}
private:
    static size_t count = 0;
};
```

**How to make safe using TM?**

# An Interesting

```
class Id
{
public:
   Id(size_
private:
   size_t const id_;
};

class Account : public Id
{
public:
   Account() : Id(count++) {}
private:
   static size_t count = 0;
};
```

**How to make safe using TM?**

- id_ const mem
- count is static (shared memory)

# An Interesting

**How to make safe using TM?**

```
class Id
{
public:
   Id(size_
private:
   size_t const id_;
};

class Account : public Id
{
public:
   Account() : Id(count++) {}
private:
   static size_t count = 0;
};
```

- id_ const mem
- count is static (shared memory)

- TBoost.STM **cannot** handle this

# C++ TM Spec **Can** Handle

# C++ TM Spec Can Handle

```
class Id
{
public:
   Id(size_t id) : id_(id) {}
private:
   size_t const id_;
};

class Account : public Id
{
public:
   // member initialization atomic / isolated
   Account() __transaction : Id(count++) { ... }
private:
   static size_t count = 0;
};
```

# C++ TM Spec Can Handle

```
class Id
{
public:
   Id(size_t id) : id_(id) {}
private:
   size_t const id_;
};

class Account : public Id
{
public:
   // member initialization atomic / isolated
   Account() __transaction : Id(count++) { ... }
private:
   static size_t count = 0;
};
```

When I first saw this, the only word that came to mind was

"Wow!"

# Optimizing Atomicity

# Optimizing Atomicity

```
class Object
{
public:
    // initialization atomic/isolated
    Object() __transaction :
        arr_(alloc_.allocate(someSize)) { ... }

    // initialization & assignment atomic/isolated
    Object(Object const &rhs) __transaction :
        arr_(alloc_.allocate(rhs.arr_, rhs.size_)) {}

private:
    size_t *arr_;
    size_t size_;
    static Allocator<size_t> alloc_;
};
```

# Optimizing Atomicity

```
class Object
{
public:
    // initialization atomic/isolated
    Object() __transaction :
        arr_(alloc_.allocate(someSize)) {

    // initialization & assignment atomi
    Object(Object const &rhs) __transaction :
        arr_(alloc_.allocate(rhs.arr_, rhs.size_)) {}

private:
    size_t *arr_;
    size_t size_;
    static Allocator<size_t> alloc_;
};
```

Try doing this with std::mutex



Wednesday, May 18, 2011

# Optimizing Atomicity

```
class Object
{
public:
    // initialization atomic/isolated
    Object() __transaction :
        arr_(alloc_.allocate(someSize)) {

    // initialization & assignment atomi
    Object(Object const &rhs) __transaction :
        arr_(alloc_.allocate(rhs.arr_, rhs.size_)) {}
```

Try doing this with std::mute x

**Disclaimer: it can be done. TBoost.STM does it.**

**Challenging to write correctly and efficiently!**

# Optimizing Atomicity

```
class Object
{
public:
    // initialization atomic/isolated
    Object() __transaction :
        arr_(alloc_.allocate(someSize)) {

    // initialization & assignment atomi
    Object(Object const &rhs) __transaction :
        arr_(alloc_.allocate(rhs.arr_, rhs.size_)) {}
```

Try doing this with std::mutex

**Disclaimer: it can be done. TBoost.STM does it.**

**Challenging to write correctly and efficiently!**

# A Simple Example

# A Simple Example

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

# A Simple Example

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

**Shared access: x, y, z.**

**How to make safe using TM?**

# Using TBoost.STM

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

# Using TBoost.STM

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

```
Obj x, y, z;

void foo()
{
   atomic(t)
   {
      Obj tmp = x * y / z;

      // access tmp
   }
}
```

# Using TBoost.STM

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

```
Obj x, y, z;

void foo()
{
   atomic(t)
   {
      Obj tmp = x * y / z;

      // access tmp
   }
}
```

**OK, but can cost performance (long tx).**

# Using TBoost.STM

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

# Using TBoost.STM

```
Obj x, y, z;

void foo()
{
    Obj tmp = x * y / z;

    // access tmp
}
```

```
Obj x, y, z;

void foo()
{
    Obj tmp;

    atomic(t)
    {
        tmp = x * y / z;
    }

    // access tmp
}
```

# Using TBoost.STM

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

```
Obj x, y, z;

void foo()
{
   Obj tmp;

   atomic(t)
   {
      tmp = x * y / z;
   }

   // access tmp
}
```

**OK, but changes behavior and suffers double assignment penalty.**

# Using TBoost.STM

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

# Using TBoost.STM

```
Obj x, y, z;

void foo()
{
    Obj tmp = x * y / z;

    // access tmp
}
```

```
Obj x, y, z;

void foo()
{
    Obj *tmp;

    atomic(t)
    {
        tmp = new Obj(x * y / z);
    }

    // access tmp
    delete tmp;
}
```

# Using TBoost.STM

```
Obj x, y, z;

void foo()
{
    Obj tmp = x * y / z;

    // access tmp
}
```

```
Obj x, y, z;

void foo()
{
    Obj *tmp;

    atomic(t)
    {
        tmp = new Obj(x * y / z);
    }

    // access tmp
    delete tmp;
```

**OK, but heap (de) allocation may be slow.**

# Using Transaction

# Using Transaction

```
Obj x, y, z;

void foo()
{
   Obj tmp = x * y / z;

   // access tmp
}
```

# Using Transaction

```
Obj x, y, z;

void foo()
{
    Obj tmp = x * y / z;

    // a
}
```

```
Obj x, y, z;

void foo()
{
    Obj tmp = __transaction ( x * y / z );

    // access tmp
}
```

# Using Transaction

```
Obj x, y, z;

void foo()
{
    Obj tmp = x * y / z;

    // a
}
```

```
Obj x, y, z;

void foo()
{
    Obj tmp = __transaction ( x * y / z );

    // access tmp
}
```

**Yes! This is exactly what we want.**

# Using Transaction

```
Obj x, y, z;

void foo()
{
    Obj tmp = x * y / z;

    // a
}
```

```
Obj x, y, z;

void foo()
{
    Obj tmp = __transac

    // access tmp
}
```

**Note: Assignment outside of tx.**

**Yes! This is exactly what we want.**

# What About

# What About

```
Obj x, y, z;

void foo()
{
    Obj tmp =
        __transaction ( x * y / z );

    // access tmp
}
```

# What About

```
Obj x, y, z;

void foo()
{
    Obj tmp =
        __transaction ( x * y / z );

    // access tmp
}
```

```
Obj x, y, z;

void foo()
{
    Obj tmp;

    __transaction
    {
        tmp = x;
        tmp *= y;
        tmp /= z;
    }

    // access tmp
    delete tmp;
}
```

# What About

```
Obj x, y, z;

void foo()
{
    Obj tmp =
        __transaction ( x * y / z );

    // access tmp
}
```

```
Obj x, y, z;

void foo()
{
    Obj tmp;

    __transaction
    {
        tmp = x;
        tmp *= y;
        tmp /= z;
    }

    // access tmp
    delete tmp;
}
```

**Two points:**

**Programmability**

**Rvalue references**

● G

# I/O: No Synchronization

# I/O: No Synchronization

```
void foo()
{
  cout << "Hello Concurrent Programming World!" << endl;
}
```

# I/O: No Synchronization

```
void foo()
{
  cout << "Hello Concurrent Programming World!" << endl;
}
```

```
// Thread 1
foo();
```

```
// Thread 2
foo();
```

# I/O: No Synchronization

```
void foo()
{
   cout << "Hello Concurrent Programming World!" << endl;
}
```

```
// Thread 1
foo();
```

```
// Thread 2
foo();
```

```
Hello Concurrent Programming World!
Hello Concurrent Programming World!
```

# I/O: No Synchronization

```
void foo()
{
  cout << “Hello Concurrent Programming World!” << endl;
}
```

```
// Thread 1
foo();
```

```
// Thread 2
foo();
```

```
Hello Concurrent Programming World!
H
```

```
Hello Hello Concurrent Concurrent Programming
Programming World! World!
```

# I/O: No Synchronization

```
void foo()
{
   cout << "Hello Concurrent Programming World!" << endl;
}
```

```
// Thread 1
foo();
```

```
// Thread 2
foo();
```

```
Hello Concurrent Programming World!
H
```

```
Hello Hello Concurrent Concurrent Programming
Programming World! World!
```

...*Hello Concurrent Programming Hell World!*...
(and other fun [and appropriate] variations)

# I/O: Transactions

# I/O: Transactions

```
void foo()
{
  cout << "Hello Concurrent Programming World!" << endl;
}
```

# I/O: Transactions

```
void foo()
{
  cout << "Hello Concurrent Programming World!" << endl;
}
```

```
// Thread 1
__transaction
{
   foo();
}
```

```
// Thread 2
__transaction
{
   foo();
}
```

# I/O: Transactions

```
void foo()
{
  cout << "Hello Concurrent Programming World!" << endl;
}
```

```
// Thread 1
__transaction
{
   foo();
}
```

```
// Thread 2
__transaction
{
   foo();
}
```

```
Hello Hello ... Hello
```

# I/O: Transactions

```
void foo()
{
  cout << "Hello Concurrent Programming World!" << endl;
}
```

```
// Thread 1
__transaction
{
   foo();
}
```

```
// Thread 2
__transaction
{
   foo();
}
```

```
Hello Hello ... Hello
```

**Three Hello's?**
**There are only two calls?**

# I/O (or Irrevocable Actions): Relaxed Transactions

# I/O (or Irrevocable Actions): Relaxed Transactions

```
void foo()
{
  cout << “Hello Concurrent Programming World!” << endl;
}
```

# I/O (or Irrevocable Actions): Relaxed Transactions

```
void foo()
{
  cout << "Hello Concurrent Programming World!" << endl;
}
```

```
// Thread 1
__transaction [[relaxed]]
{
   foo();
}
```

```
// Thread 2
__transaction [[relaxed]]
{
   foo();
}
```

# I/O (or Irrevocable Actions): Relaxed Transactions

```
void foo()
{
  cout << "Hello Concurrent Programming World!" << endl;
}
```

```
// Thread 1
__transaction [[relaxed]]
{
   foo();
}
```

```
// Thread 2
__transaction [[relaxed]]
{
   foo();
}
```

```
Hello Concurrent Programming World!
Hello Concurrent Programming World!

(only possible answer)
```

# Existing Lock-Based

# Existing Lock-Based

```
class BankAccount
{
public:
   // assume lock-based
   void withdraw(int amt);
   void deposit(int amt);
};

void transfer(int amt)
{
   __transaction [[relaxed]]
   {
      chk.withdraw(amt);
      sav.deposit(amt);
   }
}
```

# Existing Lock-Based

- Prevents interference from other txes

```
class BankAccount
{
public:
    // assume lock-based
    void withdraw(int amt);
    void deposit(int amt);
};

void transfer(int amt)
{
    __transaction [[relaxed]]
    {
        chk.withdraw(amt);
        sav.deposit(amt);
    }
}
```

# Existing Lock-Based

- Prevents interference from other txes
- Does **not** make transfer atomic

```
class BankAccount
{
public:
    // assume lock-based
    void withdraw(int amt);
    void deposit(int amt);
};

void transfer(int amt)
{
    __transaction [[relaxed]]
    {
        chk.withdraw(amt);
        sav.deposit(amt);
    }
}
```

# Existing Lock-Based

- Prevents interference from other txes
- Does **not** make transfer atomic

```
class BankAccount
{
public:
    // assume lock-based
    void withdraw(int amt);
    void deposit(int amt);
};

void transfer(int amt)
{
    __transaction [[relaxed]]
    {
        chk.withdraw(amt);
        sav.deposit(amt);
    }
}
```

**visible to non-transactional operations** → sav.deposit(amt);

# Locks and Txes

# Locks and Txes

```
class BankAccount
{
public:
    void withdraw(int amt);
    void deposit(int amt);
    void lock(); void unlock();
};

void transfer(int amt)
{
    __transaction [[relaxed]]
    {
        chk.lock();
        sav.lock();
        checking.withdraw(amt);
        savings.deposit(amt);
        // unlock
    }
}
```

# Locks and Txes

```
class BankAccount
{
public:
   void withdraw(int amt);
   void deposit(int amt);
   void lock(); void unlock();
};

void transfer(int amt)
{
   __transaction [[relaxed]]
   {
      chk.lock();
      sav.lock();
      checking.withdraw(amt);
      savings.deposit(amt);
      // unlock
   }
}
```

- Prevents transaction interference

# Locks and Txes

```
class BankAccount
{
public:
   void withdraw(int amt);
   void deposit(int amt);
   void lock(); void unlock();
};

void transfer(int amt)
{
   __transaction [[relaxed]]
   {
      chk.lock();
      sav.lock();
      checking.withdraw(amt);
      savings.deposit(amt);
      // unlock
   }
}
```

- Prevents transaction interference
- Prevents lock interference

# Locks and Txes

```
class BankAccount
{
public:
    void withdraw(int amt);
    void deposit(int amt);
    void lock(); void unlock();
};

void transfer(int amt)
{
    __transaction [[relaxed]]
    {
        chk.lock();
        sav.lock();
        checking.withdraw(amt);
        savings.deposit(amt);
        // unlock
    }
}
```

- Prevents transaction interference
- Prevents lock interference

**atomic with respect to txes and locks**

# Why Not Relax, By Default?

# Why Not Relax, By Default?

- Relaxed transactions may execute serially (isolated)

# Why Not Relax, By Default?

- Relaxed transactions may execute serially (isolated)

# Why Not Relax, By Default?

- Relaxed transactions may execute serially (isolated)
- Can degrade performance

# Why Not Relax, By Default?

- Relaxed transactions may execute serially (isolated)

- Can degrade performance

# Why Not Relax, By Default?

- Relaxed transactions may execute serially (isolated)
- Can degrade performance
- But … is there any argument for defaulting to relaxed transactions?

# But Couldn't TBoost.STM ...

# But Couldn't TBoost.STM ...

- Yes, it could
  - Transactions + Locks
  - Atomic and isolated

# But Couldn't TBoost.STM …

- Yes, it could
  - Transactions + Locks
  - Atomic and isolated

```
class BankAccount
{
public:
    void withdraw(int amt);
    void deposit(int amt);
    void lock(); void unlock();
};

void transfer(int amt)
{
    atomic(t)
    {
        t.conflict(chk.lock());
        t.conflict(sav.lock());
        chk.withdraw(amt);
        sav.deposit(amt);
    }
}
```

# But Couldn't TBoost.STM …

- Yes, it could
  - Transactions + Locks
  - Atomic and isolated
- So, why not Spec?
  - TBoost.STM's solution doesn't **generalize** well
  - Ongoing discussion
  - Proposing **tm_lock**

```
class BankAccount
{
public:
    void withdraw(int amt);
    void deposit(int amt);
    void lock(); void unlock();
};

void transfer(int amt)
{
    atomic(t)
    {
        t.conflict(chk.lock());
        t.conflict(sav.lock());
        chk.withdraw(amt);
        sav.deposit(amt);
    }
}
```

# As an Author of

# As an Author of

- Important notes:
  - TBoost.STM limited to small space
    - C++ TM Spec is not
  - TBoost.STM had code bloat
    - C++ TM Spec does not
  - Simple behavior is complex
    - C++ TM Spec it isn't

# As an Author of

- Important notes:
  - TBoost.STM limited to small space
    - C++ TM Spec is not
  - TBoost.STM had code bloat
    - C++ TM Spec does not
  - Simple behavior is complex
    - C++ TM Spec it isn't
- Point:
  - C++ TM Spec handles many things elegantly

# But What About

# But What About

- Last year
  - Only throw scalar (integral) exceptions from transactions
  - Valid concern!

# But What About

- Last year
  - Only throw scalar (integral) exceptions from transactions
  - Valid concern!
- Point:
  - Restricted only when **canceling / aborting** transactions

# But What About

- Last year
  - Only throw scalar (integral) exceptions from transactions
  - Valid concern!
- Point:
  - Restricted only when **canceling / aborting** transactions

```
void foo()
{
    __transaction
    {
        ...
        throw <insert anything here>;
    }
}
```

# But What About

- Last year
  - Only throw scalar (integral) exceptions from transactions
  - Valid concern!
- Point:
  - Restricted only when **canceling / aborting** transactions

```
void foo()
{
    __transaction
    {
        ...
        throw <insert anything here>;
    }
}
```

Perfectly legal!

# Why Restrict Exceptions?

# Why Restrict Exceptions?

```
try
{
    __transaction
    {
        ...
        __transaction_cancel
           throw TxException(txState);
    }
}
catch (TxException &e)
{
    cout << e.state();   // CRASH!
}
```

# Why Restrict Exceptions?

```
try
{
    __transaction
    {
        ...
        __transaction_cancel
          throw TxException(txState);
    }
}
catch (TxException &e)
{
    cout << e.state();   // CRASH!
}
```

Accessing state that no longer exists.

# Why Restrict Exceptions?

```
try
{
    __transaction
    {
        ...
        __transaction_cance
          throw TxException(txstate);
    }
}
catch (TxException &e)
{
    cout << e.state();  // CRASH!
}
```

Is there a better solution?
Let's find one together.

Accessing state that no longer exists.

# Summary of the C++ TM

# C++ TM Specification

# C++ TM Specification

- Goals
  - Unified C++ compiler support for TM
    - IBM, Intel, HP, Oracle, Red Hat
  - Standard C++ integration

# Achievements

# Achievements

- Baseline TM characteristics:
  - atomicity, isolation, composition

# Achievements

- Baseline TM characteristics:
  - atomicity, isolation, composition
- Integrates with C++0x memory model

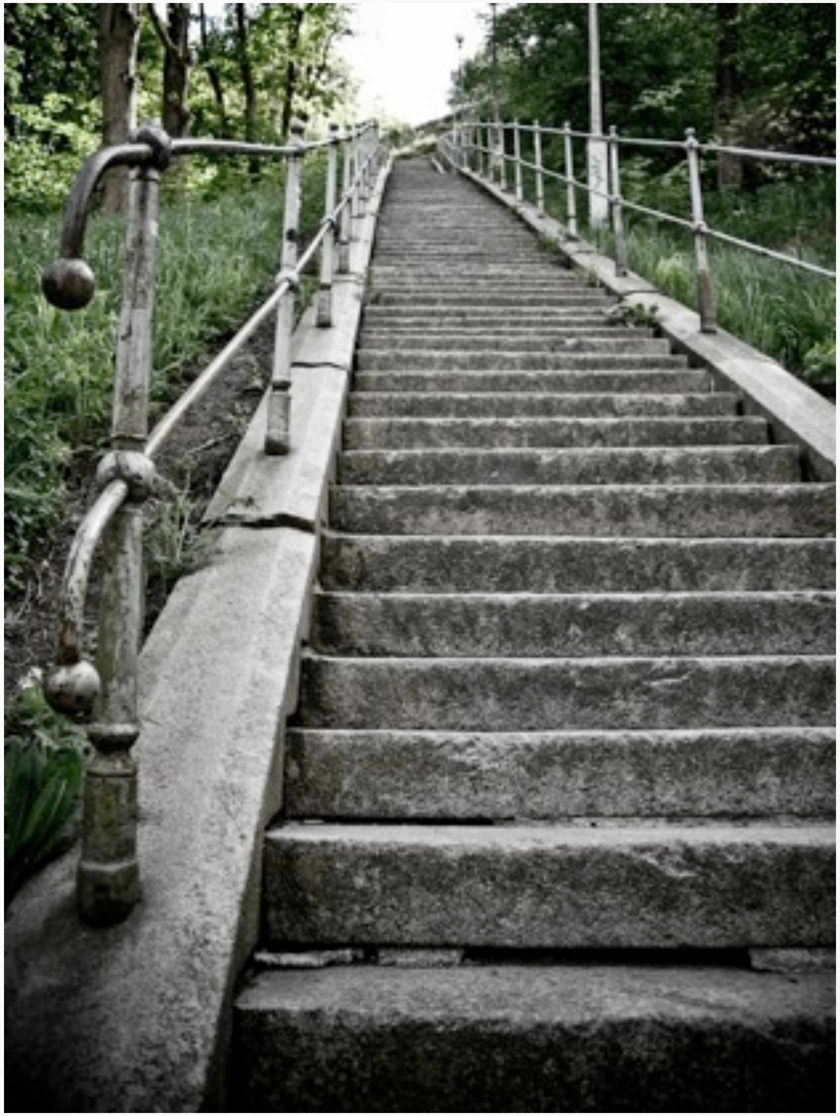
# Achievements

- Baseline TM characteristics:
  - atomicity, isolation, composition
- Integrates with C++0x memory model
- Supports important corner cases

# Achievements

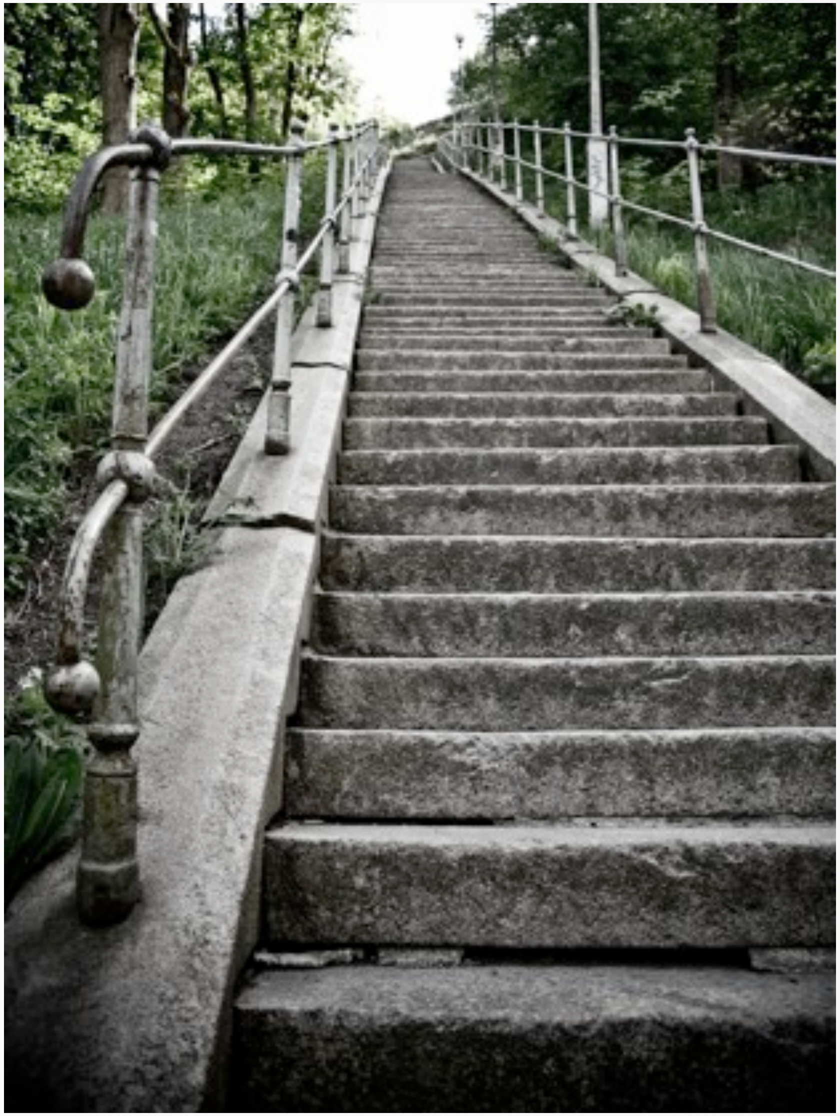
- Baseline TM characteristics:
  - atomicity, isolation, composition
- Integrates with C++0x memory model
- Supports important corner cases
- Supports I/O and irrevocable actions

# Open / Future Work



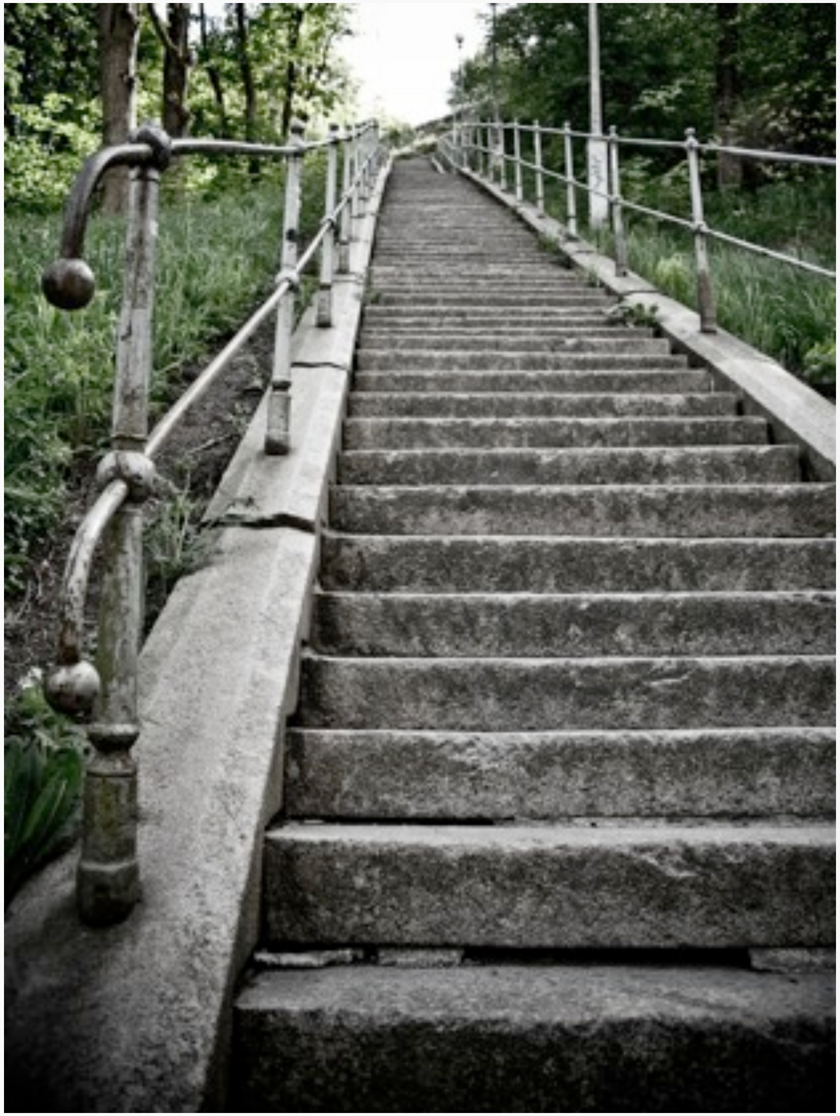
# Open / Future Work

- Exception handling model



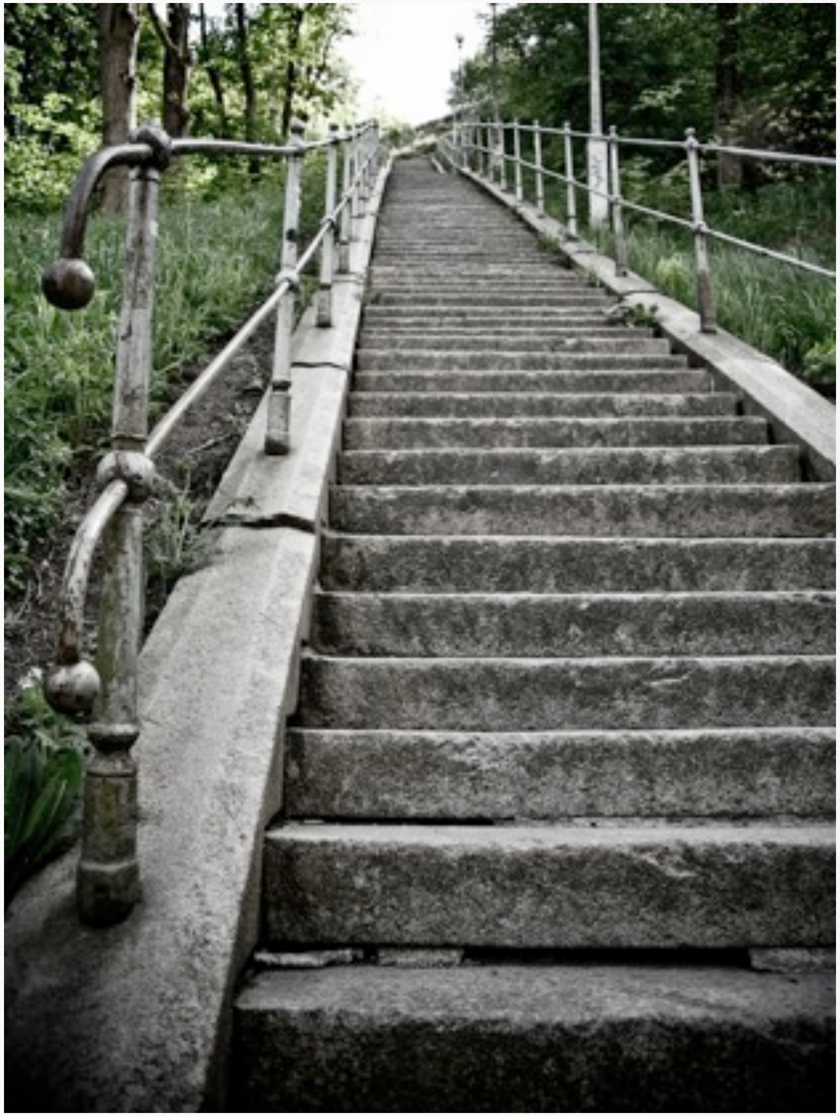
# Open / Future Work

- Exception handling model



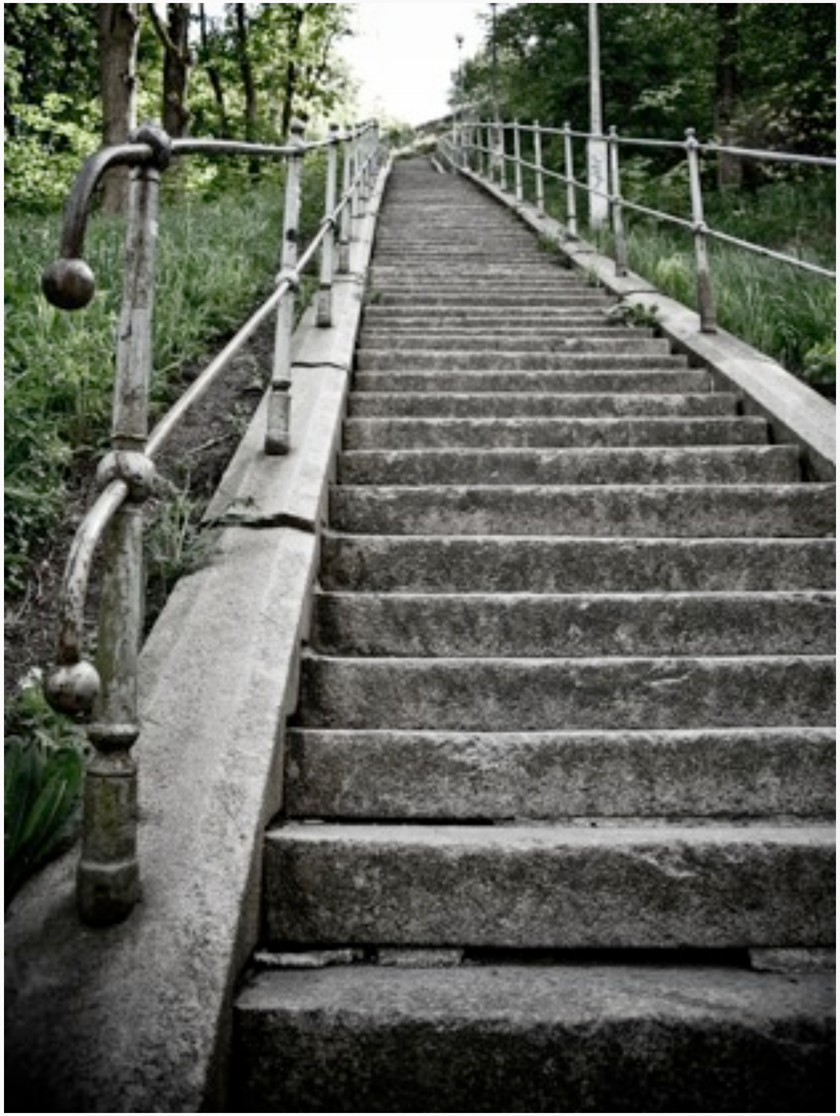
# Open / Future Work

- Exception handling model
- Transaction and lock interaction



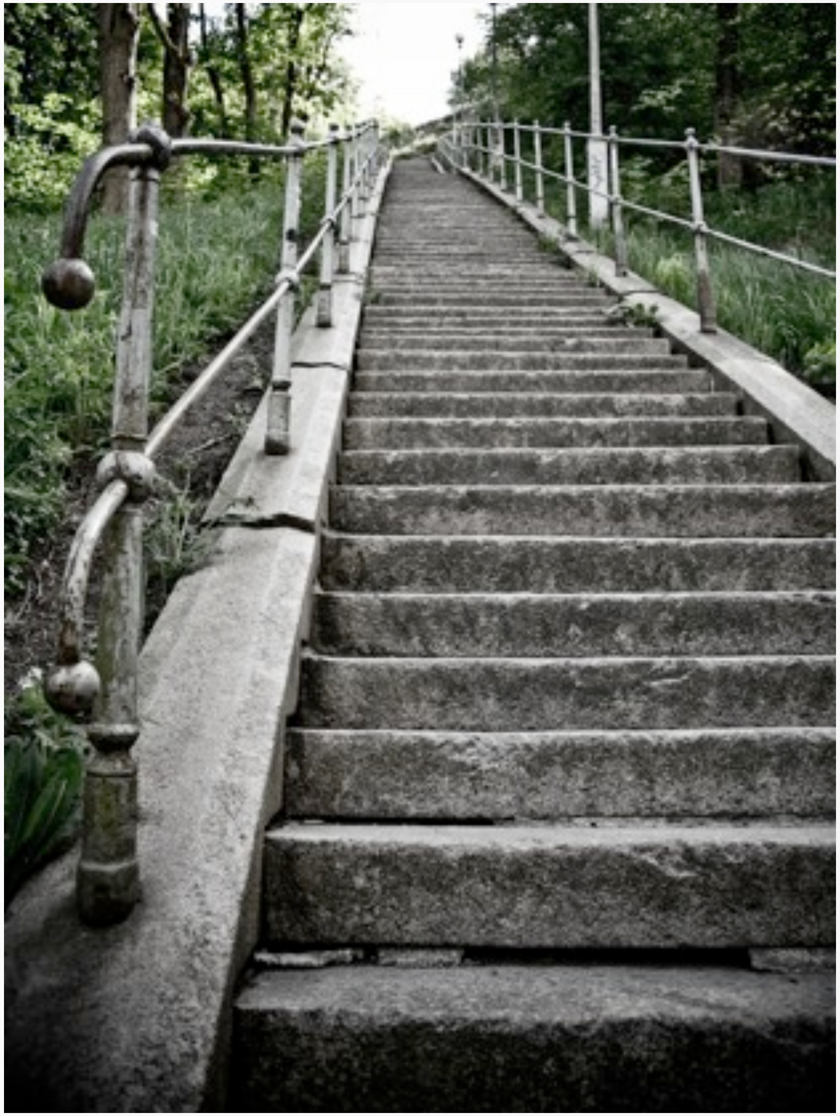
# Open / Future Work

- Exception handling model
- Transaction and lock interaction



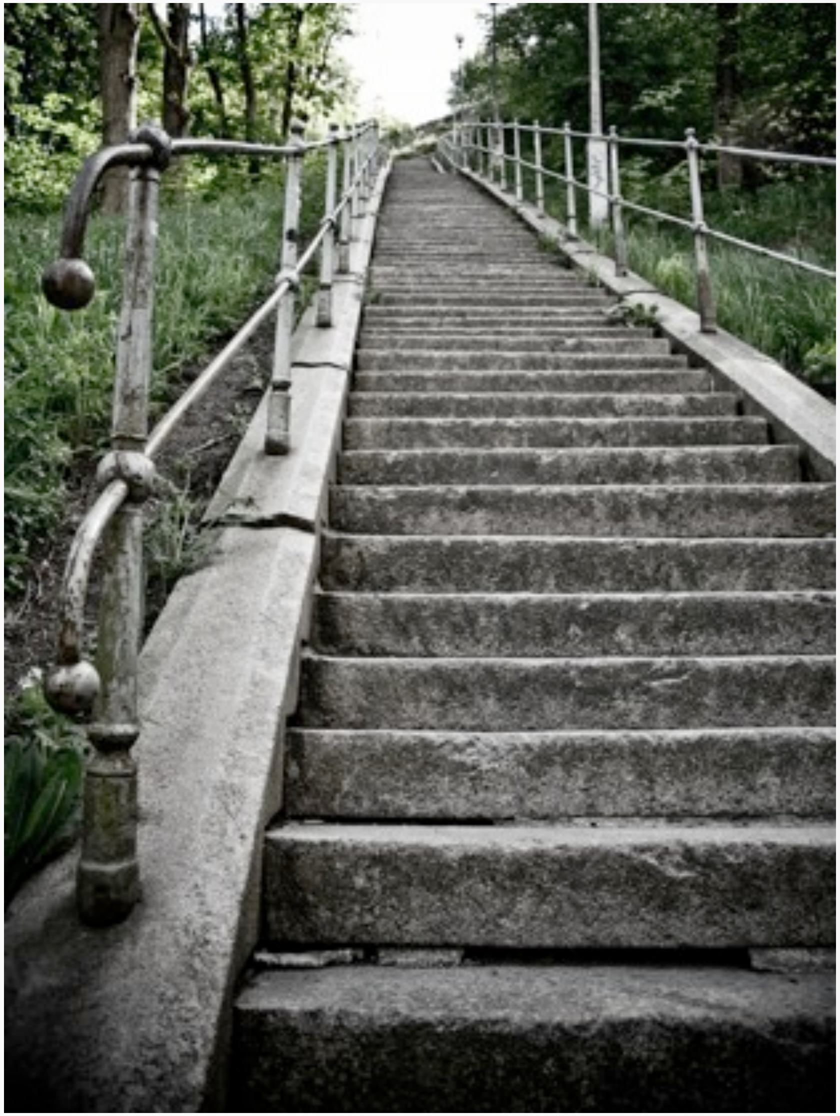
# Open / Future Work

- Exception handling model
- Transaction and lock interaction
- Dynamic errors



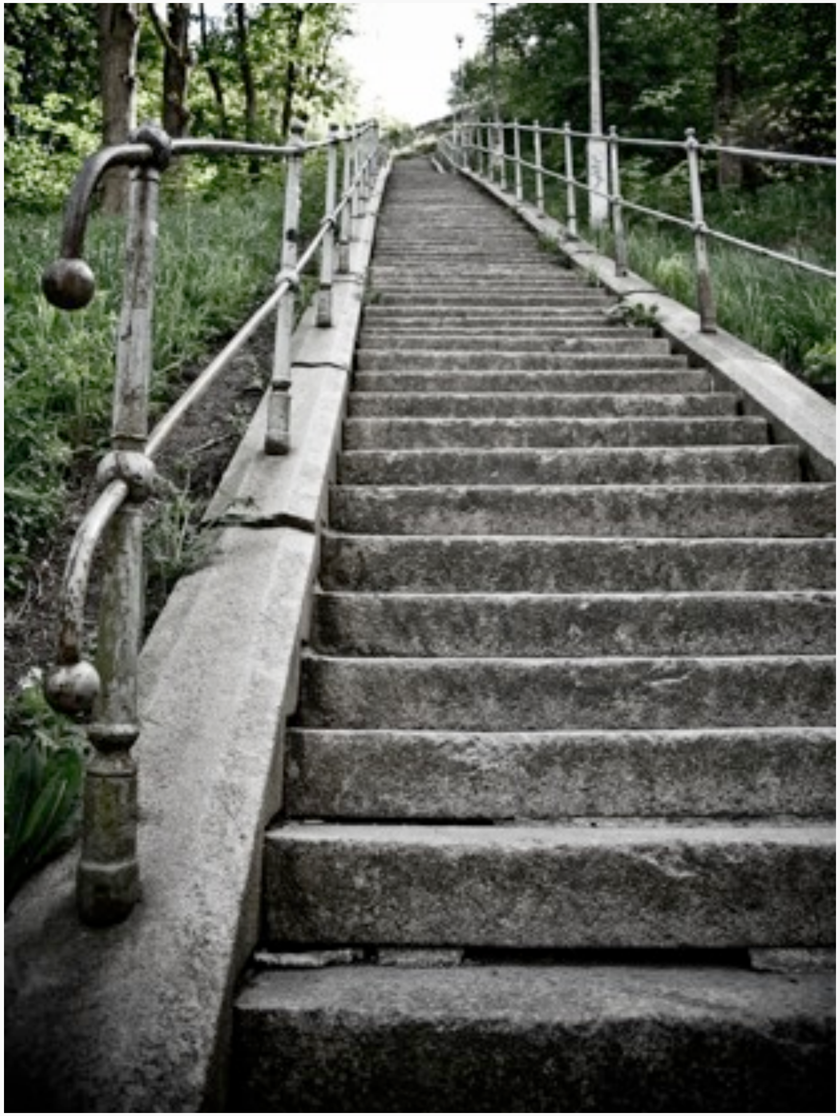
# Open / Future Work

- Exception handling model
- Transaction and lock interaction
- Dynamic errors



# Open / Future Work

- Exception handling model
- Transaction and lock interaction
- Dynamic errors
- Use cases
  - Need your feedback



# Questions? Use Cases?