LVars for distributed programming
or, LVars and CRDTs *join* forces

Lindsey Kuper
Programming Systems Lab, Intel Labs

IFIP WG 2.8 Meeting
May 26, 2015
Shared-memory parallel programming

LVars

(Observable) determinism

Distributed programming

CRDTs

(Eventual) consistency
Shared-memory parallel programming
LVars
(Observable) determinism
<table>
<thead>
<tr>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book</td>
</tr>
<tr>
<td>Shoes</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ()))))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
     async (atomicModifyIORef cart
         (\m -> (insert Book 1 m, ()))))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
            (\m -> (insert Book 1 m, ())))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ()))))
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
  async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ()))))
  async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ()))))
  res <- async (readIORef cart)
```haskell
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ()))))
    async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ()))))
    res <- async (readIORef cart)
    wait res
```
(What happens when we run this?)
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
        (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
        (\m -> (insert Shoes 1 m, ()))))
    res <- async (readIORef cart)
    wait res
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart (\m -> (insert Book 1 m, ()))))
    async (atomicModifyIORef cart (\m -> (insert Shoes 1 m, ()))))
    res <- async (readIORef cart)
    wait res
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
             (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
             (\m -> (insert Shoes 1 m, ()))))
    res <- async (readIORef cart)
    wait res

IVars: single writes, blocking (but exact) reads
[Arvind et al., 1989]

LVars: **commutative and inflationary** writes, blocking **threshold** reads
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
     (\m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
     (\m -> (insert Shoes 1 m, ())))
    res <- async (readIORef cart)
    wait res

IVars: single writes, blocking (but exact) reads
[Arvind et al., 1989]

LVars: commutative and inflationary writes, blocking threshold reads

* actually a bounded join-semilattice
data Item = Book | Shoes | ...

p :: IO (Map Item Int)
p = do cart <- newIORef empty
    async (atomicModifyIORef cart
        \m -> (insert Book 1 m, ())))
    async (atomicModifyIORef cart
        \m -> (insert Shoes 1 m, ())))
    res <- async (readIORef cart)
    wait res

IVars: single writes, blocking (but exact) reads
[Arvind et al., 1989]

LVars: **commutative and inflationary** writes, blocking **threshold** reads

* actually a bounded join-semilattice
Raises an error, since $3 \sqcup 4 = T$

```
\textbf{do}
fork (put num 3)
fork (put num 4)
```

Works fine, since $4 \sqcup 4 = 4$

```
\textbf{do}
fork (put num 4)
fork (put num 4)
```
num

\[ \begin{array}{c}
| 0 | 1 | 2 | 3 | 4 | \ldots |
\end{array} \]

\[ \begin{array}{c}
\top \\
\bot \\
\end{array} \]

\[ \text{Raises an error, since } 3 \cup 4 = \top \]
\[ \text{do} \]
\[ \text{fork (put num 3)} \]
\[ \text{fork (put num 4)} \]

\[ \text{Works fine, since } 4 \cup 4 = 4 \]
\[ \text{do} \]
\[ \text{fork (put num 4)} \]
\[ \text{fork (put num 4)} \]

\[ \text{get blocks until threshold is reached} \]
\[ \text{do} \]
\[ \text{fork (put num 4)} \]
\[ \text{get num} \]
Data structure author's obligation:

threshold set; elements must be pairwise incompatible

 Raises an error, since $3 \sqcup 4 = T$

\[
\begin{aligned}
\text{do} & \\
& \text{fork (put num 3)} \\
& \text{fork (put num 4)}
\end{aligned}
\]

 Works fine, since $4 \sqcup 4 = 4$

\[
\begin{aligned}
\text{do} & \\
& \text{fork (put num 4)} \\
& \text{fork (put num 4)}
\end{aligned}
\]

get blocks until threshold is reached

\[
\begin{aligned}
\text{do} & \\
& \text{fork (put num 4)} \\
& \text{get num}
\end{aligned}
\]
Works fine, since \texttt{incrs} commute

\begin{verbatim}
do
fork (incr1 counter)
fork (incr42 counter)
\end{verbatim}
counter

```
  T
  |
  :
  3
  2
  1
  1
```

Works fine, since `incrs commute`

```
do
  fork (incr1 counter)
  fork (incr42 counter)
```

```
get blocks until threshold is reached
do
  fork (incr1 counter)
  fork (incr42 counter)
get counter 2
```
Works fine, since $\text{incrs}$ commute

\begin{verbatim}
  \textbf{do}
  fork (incr1 counter)
  fork (incr42 counter)
\end{verbatim}

\begin{verbatim}
  \textbf{get} blocks until threshold is reached
  \textbf{do}
  fork (incr1 counter)
  fork (incr42 counter)
  \textbf{get counter} 2
\end{verbatim}

unblocks when $\text{counter}$ is at least 2
exact contents of $\text{counter}$ not observable
Distributed programming

CRDTs

(Eventual) consistency
Replication requires us to trade off between:

- **C**onsistency (all replicas agree on the data)
- **A**vailability (all replicas can read or write at all times)
- **P**artition tolerance (the system is robust to communication failure between replicas)
Replication requires us to trade off between:

- **Consistency** (all replicas agree on the data)
- **Availability** (all replicas can read or write at all times)
- **Partition tolerance** (the system is robust to communication failure between replicas)

At most two of these properties hold of a given system

[Brewer, 2000; Gilbert and Lynch, 2002]
Replication requires us to trade off between:

- **Consistency** (all replicas agree on the data)
- **Availability** (all replicas can read or write at all times)
- **Partition tolerance** (the system is robust to communication failure between replicas)

At most two of these properties hold of a given system [Brewer, 2000; Gilbert and Lynch, 2002]

In large distributed systems, **network partitions are a given**, so we have to give up one of C or A
Replication requires us to trade off between:

- **Consistency** (all replicas agree on the data)
- **Availability** (all replicas can read or write at all times)
- **Partition tolerance** (the system is robust to communication failure between replicas)

At most two of these properties hold of a given system [Brewer, 2000; Gilbert and Lynch, 2002]

In large distributed systems, **network partitions are a given**, so we have to give up one of C or A

But: **we should think of C, A, and P as more continuous than binary** [Brewer, 2012]
Replication requires us to trade off between:

- **Consistency** (all replicas agree on the data)
- **Availability** (all replicas can read or write at all times)
- **Partition tolerance** (the system is robust to communication failure between replicas)

At most two of these properties hold of a given system [Brewer, 2000; Gilbert and Lynch, 2002]

In large distributed systems, network partitions are a given, so we have to give up one of C or A

But: we should think of C, A, and P as more continuous than binary [Brewer, 2012]

We can opt for eventual consistency [Vogels, 2009]
Chord uses routing mechanisms some peer that has the desired data. A query is usually flooded through the network to find as many of P2P systems, such as Freenet and Gnutella.

3.1 higher capacity without having to upgrade all hosts at once. The next design choice is distribution must be proportional to the capabilities of the heterogeneity in the infrastructure.

Responsibilities as its peers; therefore, it can decide on the conflict resolution method that is best suited for its client’s experience. For instance, the application that maintains customer shopping carts can choose to “merge” the conflicting versions and return a single unified shopping cart.

[DeCandia et al., 2007]
Conflict-free replicated data types

[Shapiro et al., 2011]

since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client’s experience. For instance, the application that maintains customer shopping carts can choose to “merge” the conflicting versions and return a single unified shopping cart.

[DeCandia et al., 2007]
Conflict-free replicated data types

[Shapiro et al., 2011]

since the application is aware of the data schema it can decide on the conflict resolution method that is best suited for its client’s experience. For instance, the application that maintains customer shopping carts can choose to “merge” the conflicting versions and return a single unified shopping cart.

[DeCandia et al., 2007]
Two flavors of CRDTs

“Convergent”
“state-based”
CvRDTs

“Commutative”
“operation-based”
CmRDTs

\[
\text{put} \cdot \text{put} = \text{put} \cdot \text{put}
\]
Two flavors of CRDTs

“Convergent”
“state-based”
CvRDTs

⇔

“Commutative”
“operation-based”
CmRDTs

[Shapiro et al., 2011]

put · put = put · put
<table>
<thead>
<tr>
<th>LVars vs. CvRDTs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Threshold reads</strong> (deterministic)</td>
</tr>
<tr>
<td>Least-upper-bound writes (every write computes a lub)</td>
</tr>
<tr>
<td>Shared</td>
</tr>
</tbody>
</table>
Generalize LVars to inflationary, commutative writes
This gives us non-idempotent, incrementable counters (we were using them anyway…)

Extend CvRDTs with threshold queries
Systems in the wild (e.g., Amazon SimpleDB) already allow consistency choices at per-read granularity

LVars vs. CvRDTs

<table>
<thead>
<tr>
<th>Threshold reads (deterministic)</th>
<th>Ordinary reads (non-deterministic)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least-upper-bound writes (every write computes a lub)</td>
<td>Inflationary, commutative writes (only replica merges must be lubs)</td>
</tr>
</tbody>
</table>

So, to join forces:
Generalize LVars to inflationary, commutative writes. This gives us non-idempotent, *incrementable* counters (we were using them anyway…)

Extend CvRDTs with threshold queries. Systems in the wild (e.g., Amazon SimpleDB) already allow consistency choices at per-read granularity.
Deterministic **threshold queries** of CvRDTs:

Block only until a threshold element appears at **one** replica (that’s all we need!)
Programming Systems Lab
(We’re hiring! Email me!)