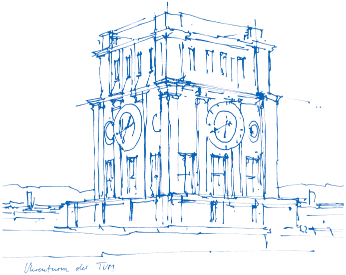
No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System

Dominik Durner, Thomas Neumann

April 10, 2019

Technische Universität München

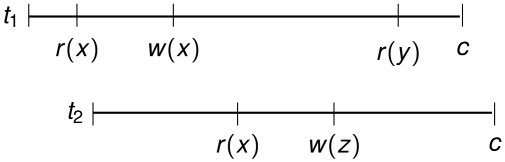


Motivation

- Concurrency control schemes only approximate the class of serializable schedules, such as 2PL, OCC, TicToc
- Therefore, unexpected behavior and also unnecessary aborts are introduced
- Spurious aborts due to implementation artifacts that are hard to understand

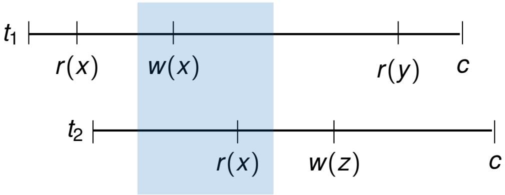
Motivation

- Concurrency control schemes only approximate the class of serializable schedules, such as 2PL, OCC, TicToc
- Therefore, unexpected behavior and also unnecessary aborts are introduced
- Spurious aborts due to implementation artifacts that are hard to understand
- For example, 2PL cannot accept:



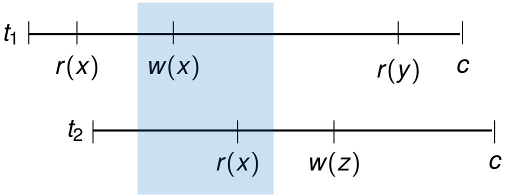
Motivation

- Concurrency control schemes only approximate the class of serializable schedules, such as 2PL, OCC, TicToc
- Therefore, unexpected behavior and also unnecessary aborts are introduced
- Spurious aborts due to implementation artifacts that are hard to understand
- For example, 2PL cannot accept:



Motivation

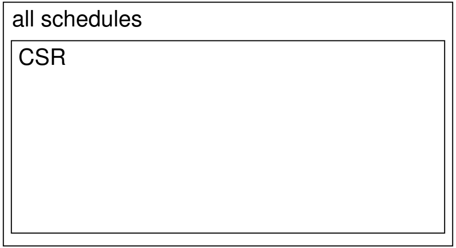
- Concurrency control schemes only approximate the class of serializable schedules, such as 2PL, OCC, TicToc
- Therefore, unexpected behavior and also unnecessary aborts are introduced
- Spurious aborts due to implementation artifacts that are hard to understand
- For example, 2PL cannot accept:



- Only Serialization Graph Testing (SGT) accepts all valid schedules
- SGT seems to be too expensive and not scalable

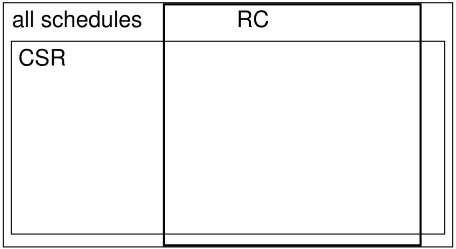
Motivation: Desired Schedules

- Conflict graphs allow to accept all conflict serializable schedules



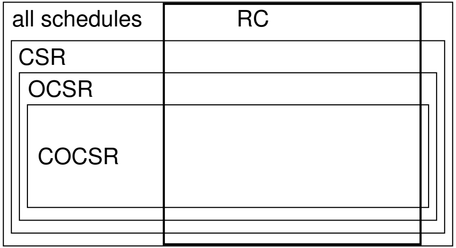
Motivation: Desired Schedules

- Conflict graphs allow to accept all conflict serializable schedules
- Recoverability is independent of serializability



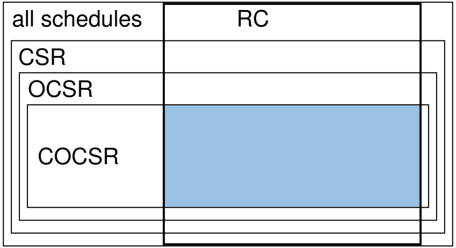
Motivation: Desired Schedules

- Conflict graphs allow to accept all conflict serializable schedules
- Recoverability is independent of serializability
- DBMS users expect to see committed changes



Motivation: Desired Schedules

- Conflict graphs allow to accept all conflict serializable schedules
- Recoverability is independent of serializability
- DBMS users expect to see committed changes

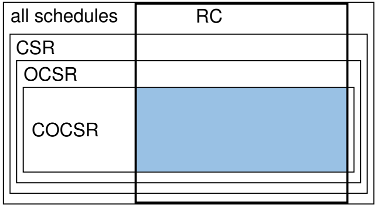


Note that $S2PL \subsetneq COCSR \cap RC$

Contribution

Our approach leverages the conflict graph and

1. accepts all useful $COCSR \cap RC$ schedules
2. meets users' expectations
3. has low overhead for maintaining the graph
4. scales to many-core systems



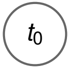
Serialization Graph Testing (SGT)

- Theorem: $s \in CSR \Leftrightarrow CG(s)$ is acylic
- Update $CG(s)$ at operation arrival and allow if $CG(s)$ is acyclic
- Remove all outgoing edges of a node at its deletion

Serialization Graph Testing (SGT)

- Theorem: $s \in CSR \Leftrightarrow CG(s)$ is acylic
- Update $CG(s)$ at operation arrival and allow if $CG(s)$ is acyclic
- Remove all outgoing edges of a node at its deletion

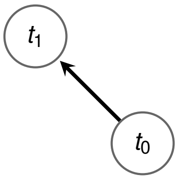
Example: $s = r_0[x]\, w_0[x]$



Serialization Graph Testing (SGT)

- Theorem: $s \in CSR \Leftrightarrow CG(s)$ is acylic
- Update $CG(s)$ at operation arrival and allow if $CG(s)$ is acyclic
- Remove all outgoing edges of a node at its deletion

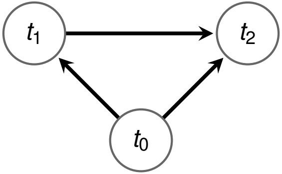
Example: $s = r_0[x]\, w_0[x]\, r_1[x]$



Serialization Graph Testing (SGT)

- Theorem: $s \in CSR \Leftrightarrow CG(s)$ is acylic
- Update $CG(s)$ at operation arrival and allow if $CG(s)$ is acyclic
- Remove all outgoing edges of a node at its deletion

Example: $s = r_0[x]\, w_0[x]\, r_1[x]\, r_2[x]\, w_2[x]$



Serialization Graph Testing (SGT)

- Theorem: $s \in CSR \Leftrightarrow CG(s)$ is acylic
- Update $CG(s)$ at operation arrival and allow if $CG(s)$ is acyclic
- Remove all outgoing edges of a node at its deletion

Example: $s = r_0[x]\ w_0[x]\ r_1[x]\ r_2[x]\ w_2[x]\ w_2[y]\ c_2\ c_0\ c_1$

$$\begin{tikzcd} t_1 \arrow[rr] & & t_2 \\ & t_0 \arrow[ul] \arrow[ur] & \end{tikzcd}$$

$\Rightarrow s \in CSR$

SGT Lacked Practical Relevance

- SGT has the best theoretical properties of accepting all valid schedules
- However, previous work fails to implement SGT efficiently in practice

SGT Lacked Practical Relevance

- SGT has the best theoretical properties of accepting all valid schedules
- However, previous work fails to implement SGT efficiently in practice

We developed the first practical and scalable algorithm that leverages
the theoretical superior concept of graph-based serialization testing

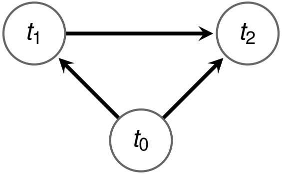
Prerequisites for Node Deletions

Pitfall: Deletion of a committed node t_c

Prerequisites for Node Deletions

Pitfall: Deletion of a committed node t_c

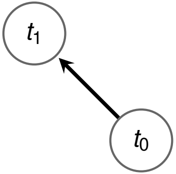
Example: $s = r_0[x]\ w_0[x]\ r_1[x]\ r_2[x]\ w_2[x]\ w_2[y]\ c_2$



Prerequisites for Node Deletions

Pitfall: Deletion of a committed node t_c

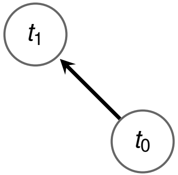
Example: $s = r_0[x]\ w_0[x]\ r_1[x]\ r_2[x]\ w_2[x]\ w_2[y]\ c_2$



Prerequisites for Node Deletions

Pitfall: Deletion of a committed node t_c

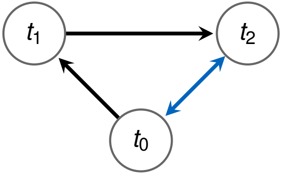
Example: $s = r_0[x]\ w_0[x]\ r_1[x]\ r_2[x]\ w_2[x]\ w_2[y]\ c_2\ \mathbf{r_0[y]}\ c_0\ c_1$



Prerequisites for Node Deletions

Pitfall: Deletion of a committed node t_c

Example: $s = r_0[x]\ w_0[x]\ r_1[x]\ r_2[x]\ w_2[x]\ w_2[y]\ c_2\ \mathbf{r_0[y]}\ c_0\ c_1$

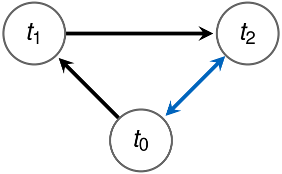


$\Rightarrow s \notin CSR$, but not detectable if t_2 was deleted

Prerequisites for Node Deletions

Pitfall: Deletion of a committed node t_c

Example: $s = r_0[x]\ w_0[x]\ r_1[x]\ r_2[x]\ w_2[x]\ w_2[y]\ c_2\ \mathbf{r_0[y]}\ c_0\ c_1$



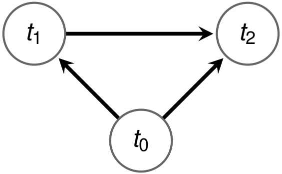
$\Rightarrow s \notin CSR$, but not detectable if t_2 was deleted

Deletion of committed node is only allowed if all incoming edges are removed

(Log-) Recoverability Constraints

Every transaction commit needs to wait until it is not dependent on in-flight results

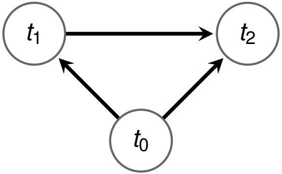
Example: $s = r_0[x]\, w_0[x]\, r_1[x]\, r_2[x]\, w_2[x]\, w_2[y]\, c_2\, c_0\, c_1$



(Log-) Recoverability Constraints

Every transaction commit needs to wait until it is not dependent on in-flight results

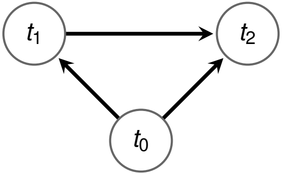
Example: $s = r_0[x]\ w_0[x]\ r_1[x]\ r_2[x]\ w_2[x]\ w_2[y]\ c_2\ \cancel{c_0}\ \cancel{c_1}\ \mathbf{a_0}\ \mathbf{a_1}$



(Log-) Recoverability Constraints

Every transaction commit needs to wait until it is not dependent on in-flight results

Example: $s = r_0[x]\, w_0[x]\, r_1[x]\, r_2[x]\, w_2[x]\, w_2[y]\, c_2\, \cancel{c_0\, c_1}\, \mathbf{a_0}\, \mathbf{a_1}$



No incoming write-read, write-write edge from an uncommitted node allowed

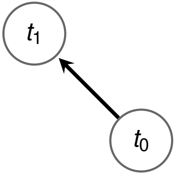
Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

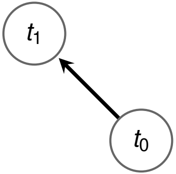
Example: $s = r_0[x]\ w_1[x]\ c_1$



Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

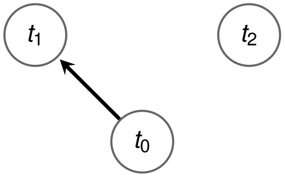
Example: $s = r_0[x]\ w_1[x]\ \cancel{c_1}\ d_1$



Preserving the Commit Order

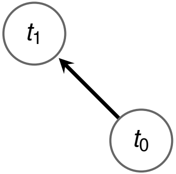
No (uncommitted) incoming edge at commit time to preserve the commit order

Example: $s = r_0[x]\ w_1[x]\ \cancel{c_1}\ d_1\ r_2[y]\ c_2$



No (uncommitted) incoming edge at commit time to preserve the commit order

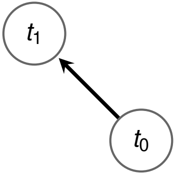
Example: $s = r_0[x]\ w_1[x]\ \cancel{c_1}\ d_1\ r_2[y]\ c_2\ w_0[y]$



Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

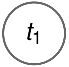
Example: $s = r_0[x]\ w_1[x]\ \cancel{c_1}\ d_1\ r_2[y]\ c_2\ w_0[y]\ c_0$



Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

Example: $s = r_0[x]\ w_1[x]\ \cancel{c_1}\ d_1\ r_2[y]\ c_2\ w_0[y]\ c_0\ c_1$



Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

Example: $s = r_0[x]\ w_1[x]\ \cancel{c_1}\ d_1\ r_2[y]\ c_2\ w_0[y]\ c_0\ c_1$

$s_{orig} = r_0[x]\ w_1[x]\ c_1\ r_2[y]\ c_2\ w_0[y]\ c_0$

with $s' = t_2\ t_0\ t_1$, but $s_{orig} \notin COCSR$

Preserving the Commit Order

No (uncommitted) incoming edge at commit time to preserve the commit order

Example: $s = r_0[x]\ w_1[x]\ \cancel{c_1}\ d_1\ r_2[y]\ c_2\ w_0[y]\ c_0\ c_1$

$$s_{orig} = r_0[x]\ w_1[x]\ c_1\ r_2[y]\ c_2\ w_0[y]\ c_0$$

with $s' = t_2\ t_0\ t_1$, but $s_{orig} \notin COCSR$

All useful $COCSR \cap RC$ schedules accepted due to commit delays

Committed nodes are deleted directly including all outgoing edges

Scaling of our SGT-based Approach

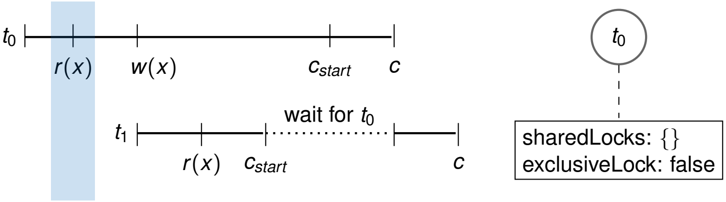
- No incoming edges to commit simplifies cycle check
- Conflict graph is accessed concurrently by multiple threads
- No other transaction is allowed to modify a node during its final check

Scaling of our SGT-based Approach

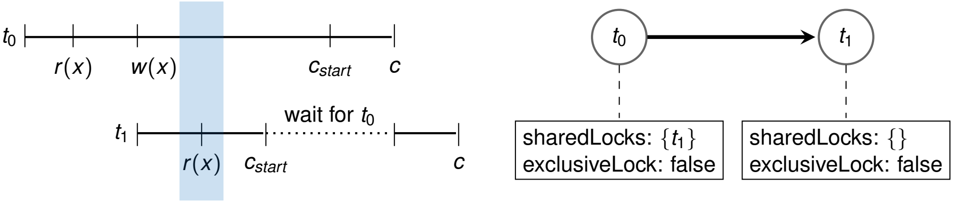
- No incoming edges to commit simplifies cycle check
- Conflict graph is accessed concurrently by multiple threads
- No other transaction is allowed to modify a node during its final check

Transaction local shared/exclusive locks help to scale the graph

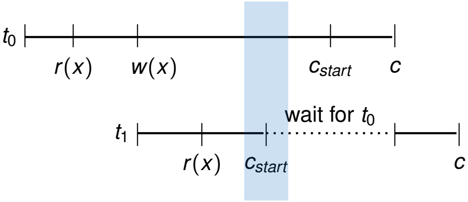
Example of our SGT-based Approach

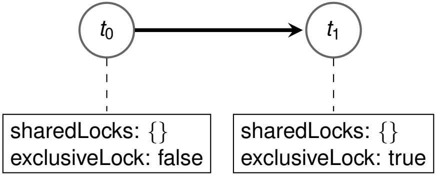


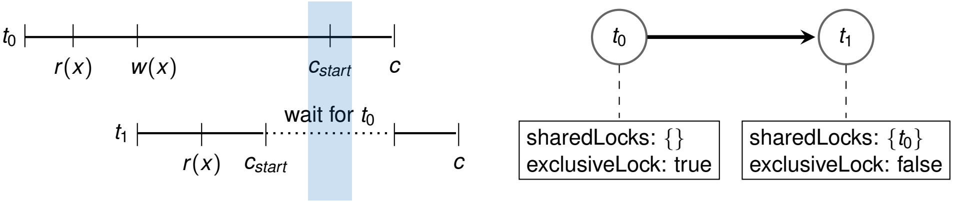
Example of our SGT-based Approach



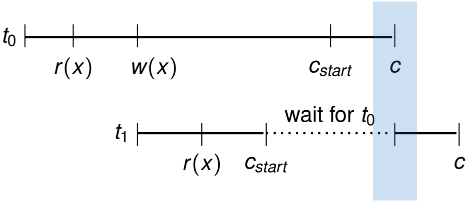
Example of our SGT-based Approach

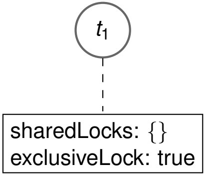





t_0
$r(x)$
$w(x)$
c_{start}
c
wait for t_0
t_1
$r(x)$
c_{start}
c
t_0
t_1
sharedLocks: $\{\}$
exclusiveLock: true
sharedLocks: $\{t_0\}$
exclusiveLock: false

Example of our SGT-based Approach





Experimental Evaluation

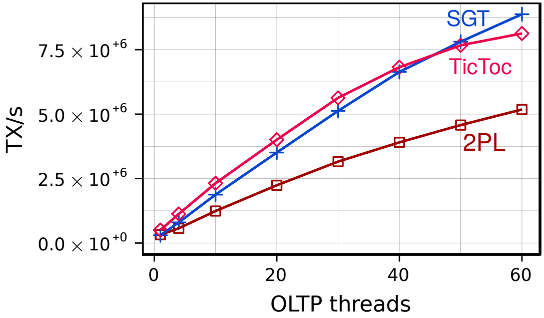
Setup:

- 4-socket Intel Xeon server (60 cores) with 1TB DRAM
- Every transaction is scheduled on one worker thread
- Aborts require undos and restarts of the aborted transactions

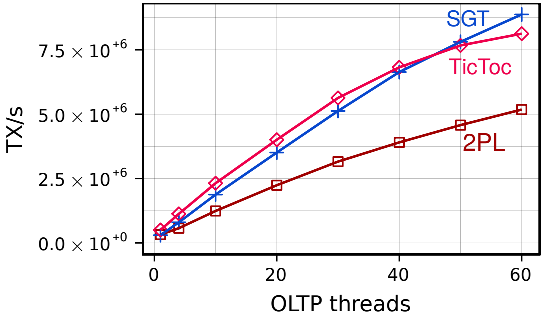
Algorithms:

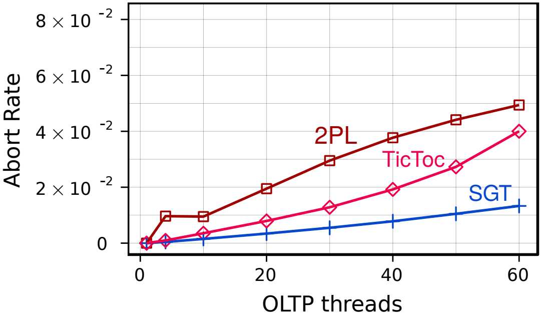
- Our SGT-based approach
- TicToc
- 2PL with row based atomic read-write locks and deadlock prevention

SmallBank Medium Contention (1000 Customers)

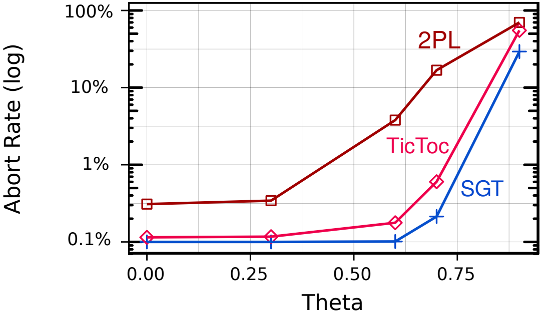


SmallBank Medium Contention (1000 Customers)

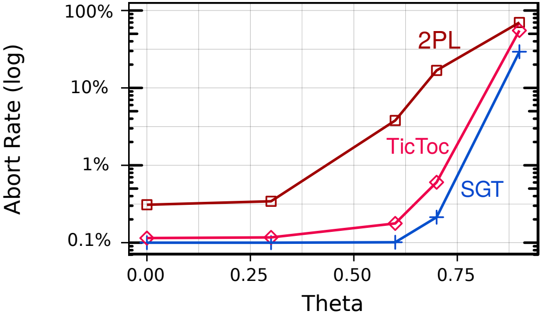


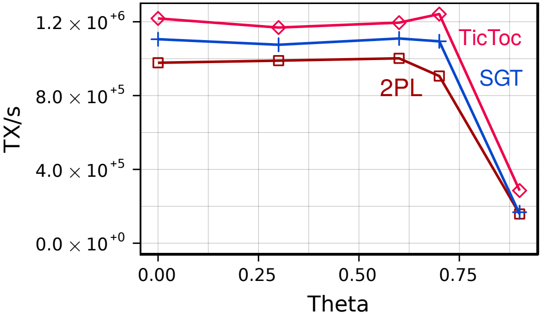


YCSB-A, 50% writes, 16 queries/tx, 60 threads



YCSB-A, 50% writes, 16 queries/tx, 60 threads

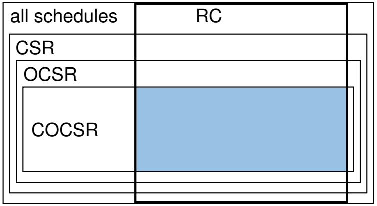




Our SGT has competitive throughput while reducing aborts significantly!

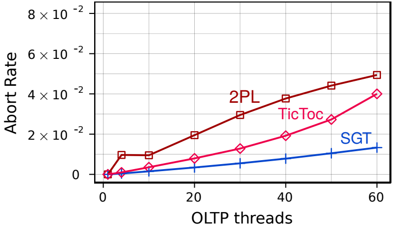
Summary: Our graph-based concurrency control algorithm

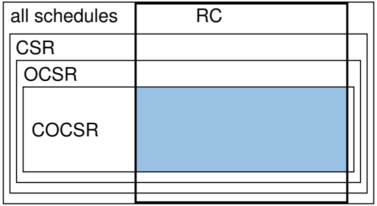
accepts all useful
COCSR ∩ *RC* schedules



Summary: Our graph-based concurrency control algorithm

accepts all useful
COCSR ∩ *RC* schedules

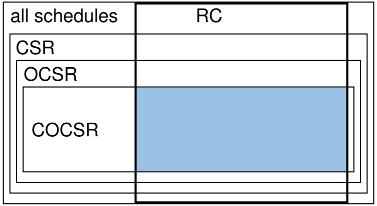


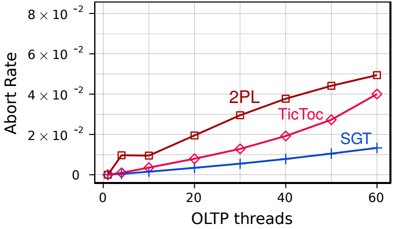


reduces aborted schedules
and meets users' expectations

Summary: Our graph-based concurrency control algorithm

accepts all useful
COCSR ∩ *RC* schedules





reduces aborted schedules
and meets users' expectations

has low protocol overhead
and scales to many-core systems