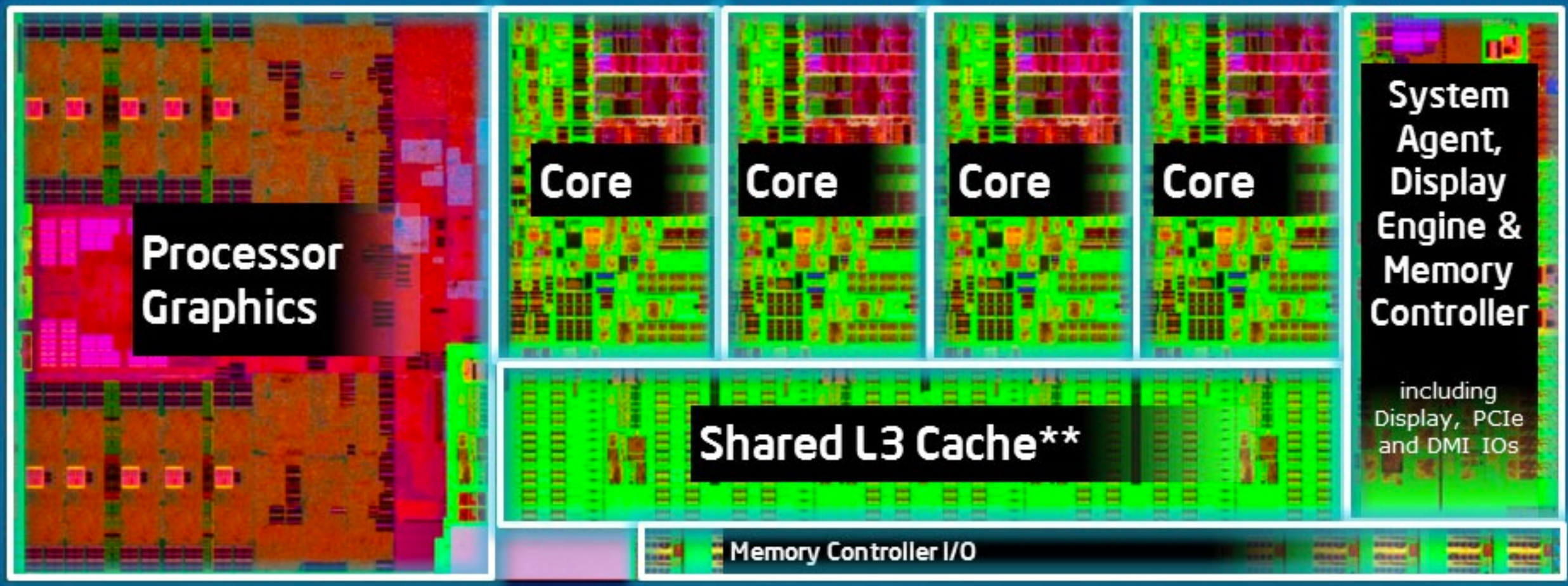
The Push/Pull model of serializability

Eric Koskinen
Research Staff Member
IBM TJ Watson Research Center

Joint work with Matthew Parkinson (Microsoft)

4th Generation Intel® Core™ Processor Die Map

22nm Haswell Tri-Gate 3-D Transistors



Quad core die shown above | Transistor count: 1.4Billion | Die size: 177mm^2

** Cache is shared across all 4 cores and processor graphics

`XBEGIN`, `XEND` and `XABORT`

`XBEGIN`, `XEND` and `XABORT`

Version 4.7

Transactions

```
x := 0; y := 0;
```

Thread 1

```
atomic {
    x := 3;
    y := 9;
}
```

Thread 2

```
atomic {
   if (x>y) {
      //shdnt happen
   } else {
      ...
}
```

Pessimism!

Thread

```
atomic {
    x := 3;
    y := 9;
}
```

x changed?

y changed?

Pessimism!

Thread

```
atomic {
    x := 3;
    y := 9;
}
```

x changed?

y changed?

Optimism!

Thread

```
atomic {
    x := 3;
    y := 9;
}
```

all ok?

Pessimism!

Serializability!

Interleaved execution equivalent to some serial execution.

Optimism!

Pessimism!

Opacity!

Interleaved execution
equivalent to some
serial execution
and
cannot observe
intermediate state.

Serializability!

Interleaved execution
equivalent to some
serial execution.

Optimism!

Pessimism!

Opacity!

Serializability!

Memory Ops

ADT Ops
(Boosting)

Optimism!

Thread

```
atomic {
    x := 3;
    y := 9;
}
```

Thread

```
atomic {
    stk.push(4);
    ht.get('a');
}
```

Pessimism!

Opacity!

Hardware TM
Software TM
Hybrid TM

Serializability!

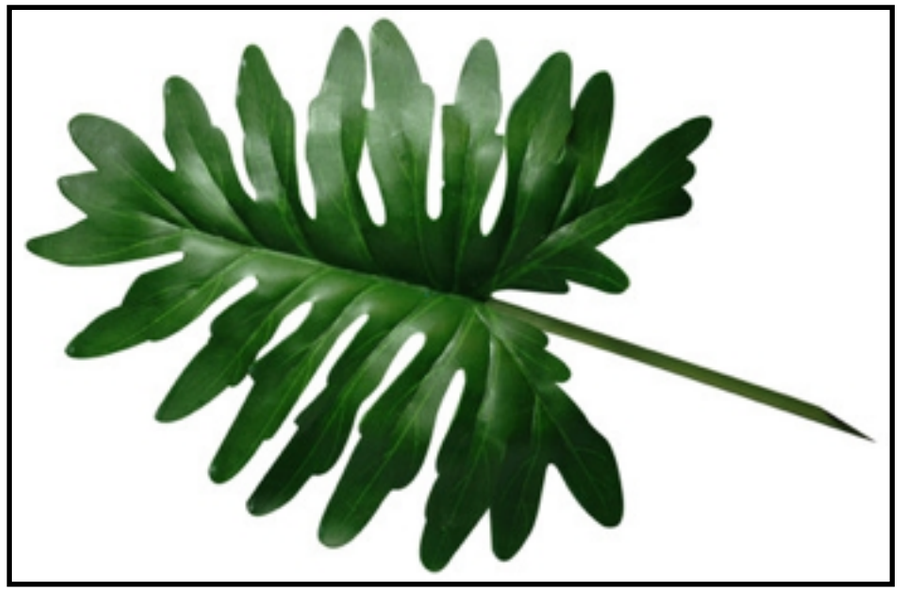
Memory Ops

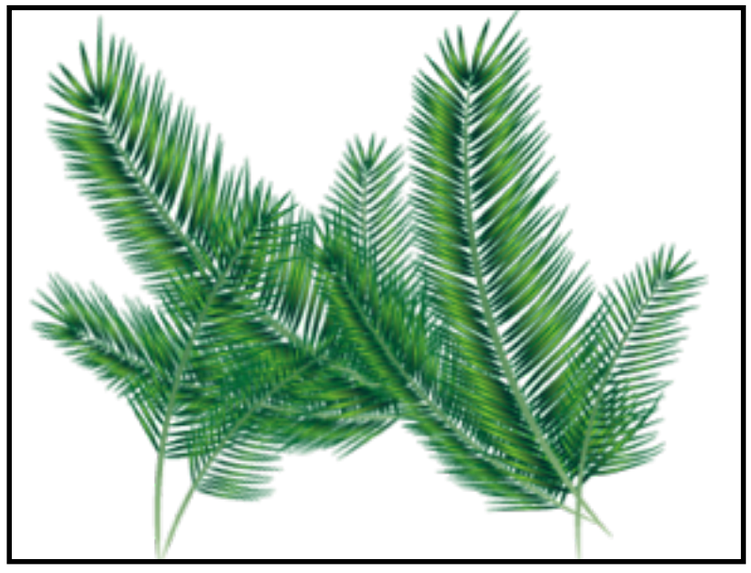
Optimism!

ADT Ops
(Boosting)

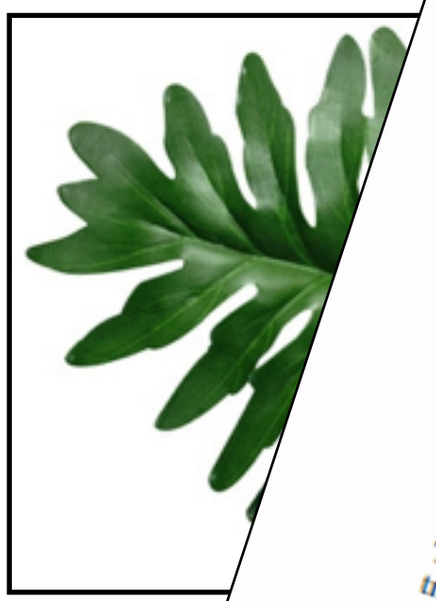

Pessimism!
Consistency
Irrevocability
Opacity!
Hardware TM
Software TM
Hybrid TM
Open Nested
Serializability!
Lazy Reads
Optimism!
Memory Ops
Eager Writes
ADT Ops
(Boosting)
Dependent Txns

Unified theory





Unified theory



The Push/Pull model of transactions

Eric Koskinen Matthew Parkinson

Abstract

We present a general theory of serializability, unifying a wide range of transactional algorithms, including some that are yet to come. To this end, we provide a compact semantics in which concurrent transactions *push* their effects into the shared view (or *unpush* to recall effects) and *pull* the effects of potentially uncommitted concurrent transactions into their local view (or *unpull* to detangle). Each operation comes with simple side-conditions given in terms of commutativity (Lipton's left-movers and right-movers [24]).

The benefit of this model is that most of the elaborate reasoning (coinduction, simulation, subtle invariants, etc.) necessary for proving the serializability of a transactional algorithm is already proved within the semantic model. Thus, proving serializability (or opacity) amounts simply to mapping the algorithm on to our rules, and showing that it satisfies the rules' side-conditions.

1. Introduction

Recent years have seen an explosion of research on methods of providing `atomic` sections in modern programming languages, typically implemented via transactional memory (TM). The `atomic` keyword provides programmers with a powerful concurrent programming building block: the ability to specify when a thread's operations on shared memory should appear to take place instantly when viewed by another thread.

To support such a construct, we must be able to reason about atomicity. Implementations typically achieve this by dynamically detecting conflicts between concurrent threads. This can be done tracking memory operations in hardware [15–17] or software [4, 6, 8, 14, 25]. Meanwhile, an alternate approach exploits abstract-level notions of conflict over linearizable data-structure operations such as commutativity [11, 20, 21, 28]. Both levels of abstraction chose between optimistic execution, pessimistic execution, and circumstances under which one tures of the two. Finally, there are multiple

Unfortunately, we lack myriad of models leads to

tions (e.g. transactional boosting [11]). Today, at best, we have two custom semantics for reasoning about the models individually, but no unified view.

We present a simple calculus that illuminates the core of transactional memory systems. In our model concurrent transactions *push* their effects into the shared log (or *unpush* to roll-back) and *pull* in the effects of potentially uncommitted concurrent transactions (or *unpull* to detangle). Moreover, transactions can push or pull operations in non-chronological orders, provided certain commutativity (Lipton left/right-movers [24]) conditions hold. The benefit of this semantic model is that most of the elaborate reasoning (coinduction, simulation relations, subtle invariants, etc.) necessary for proving the correctness of a transactional algorithm is contained within the semantic model, and need only be proved once.

Our work formulates an expressive class of TM systems including: opaque [10]: transactions *may* share their uncommitted effects. we have applied it to a wide range of transactional This choice carves out a design space for implementation memories (Intel [17], IBM [16]), pessimistic TMs [4, 11, 25], hybrid optimistic/pessimistic TMs such as irrevocability [34], open nested transactions [28], and abstract-level techniques such as boosting [11].

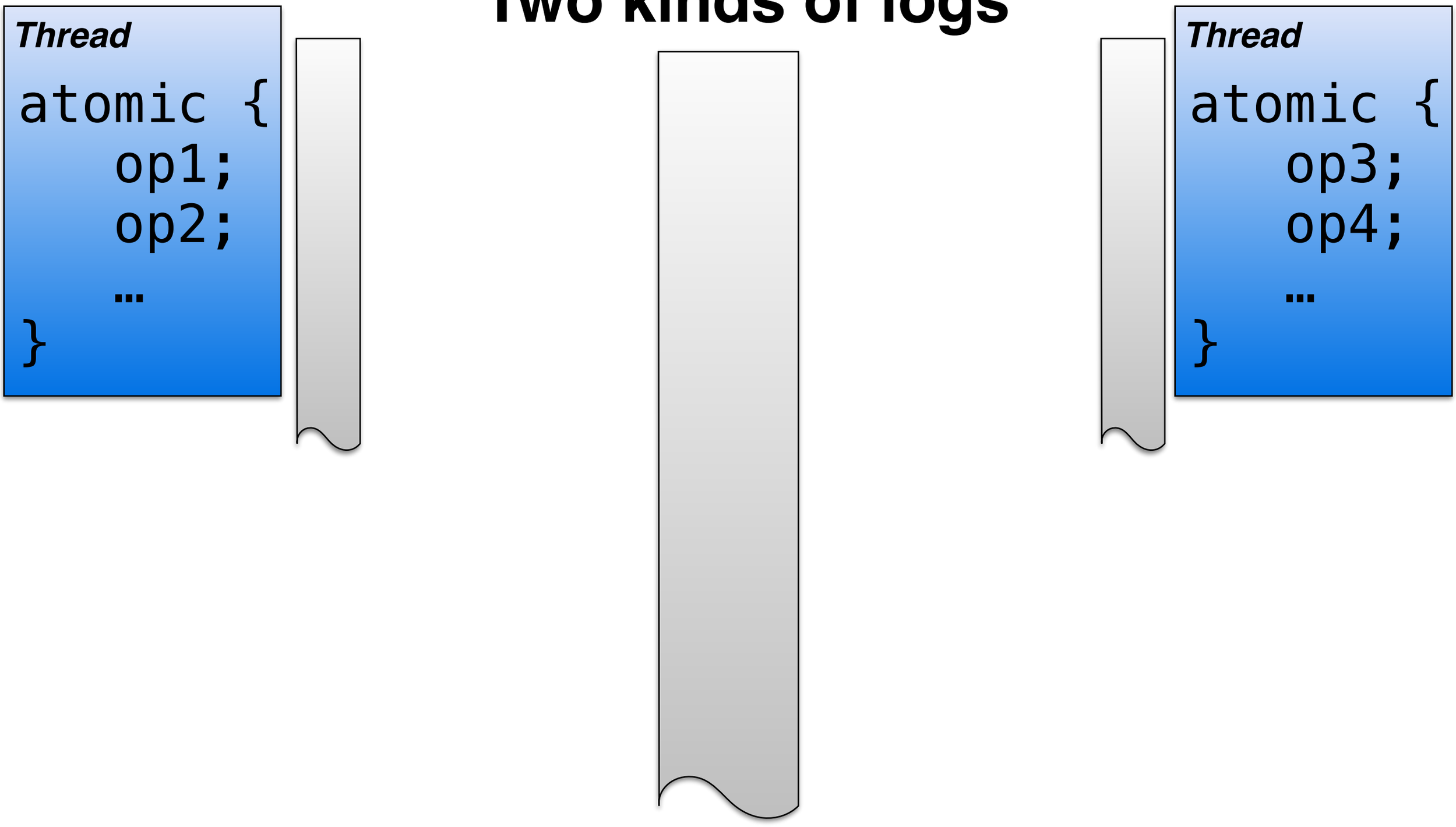
Our choice of expressiveness includes transactions that are not advantage of the full spectrum of possibilities (e.g. actions [30]) and is relatively unrestrictive ness criteria. However, despite gives the appropriate while, we

One model to rule them all.

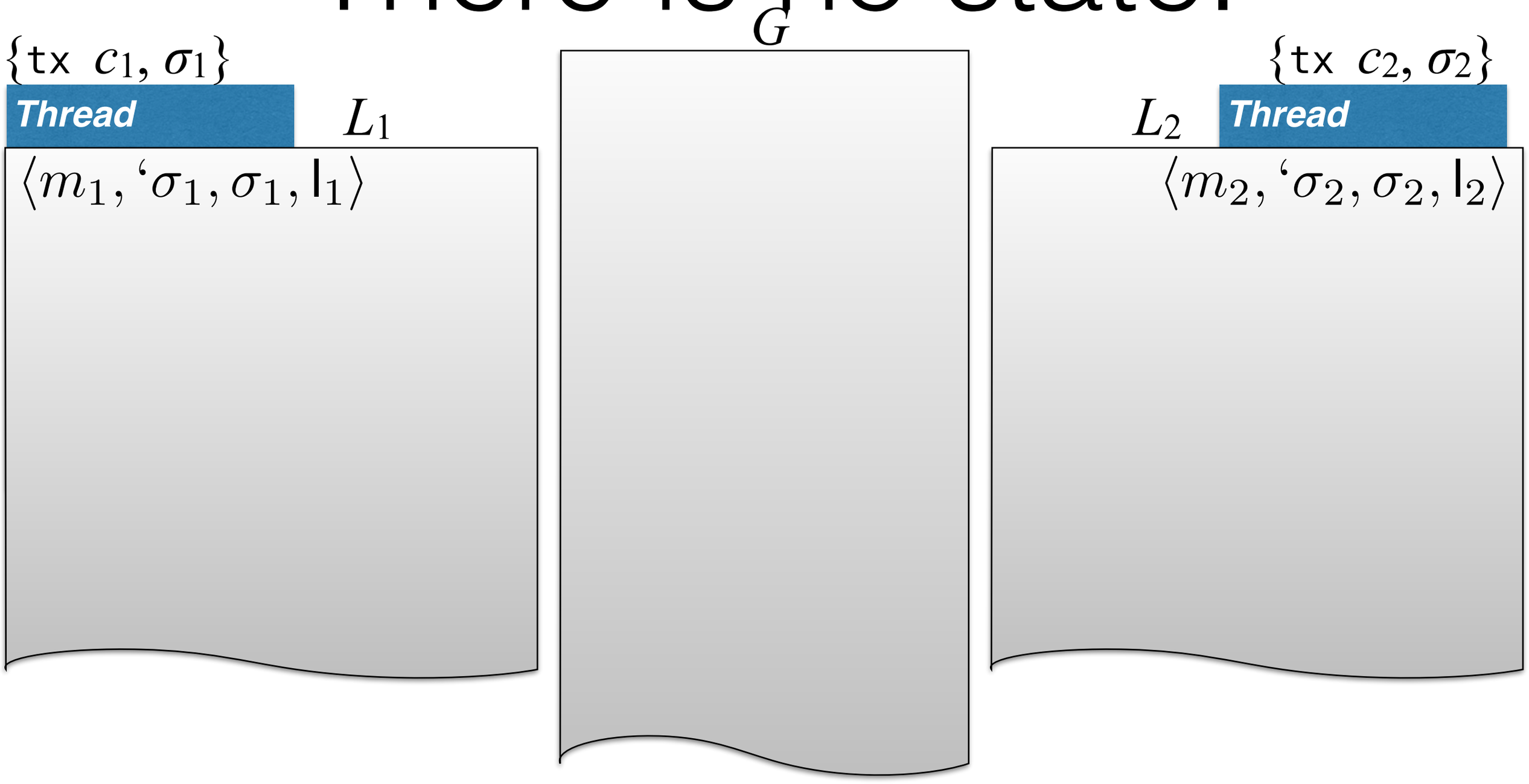
There is no state.

There is no state.

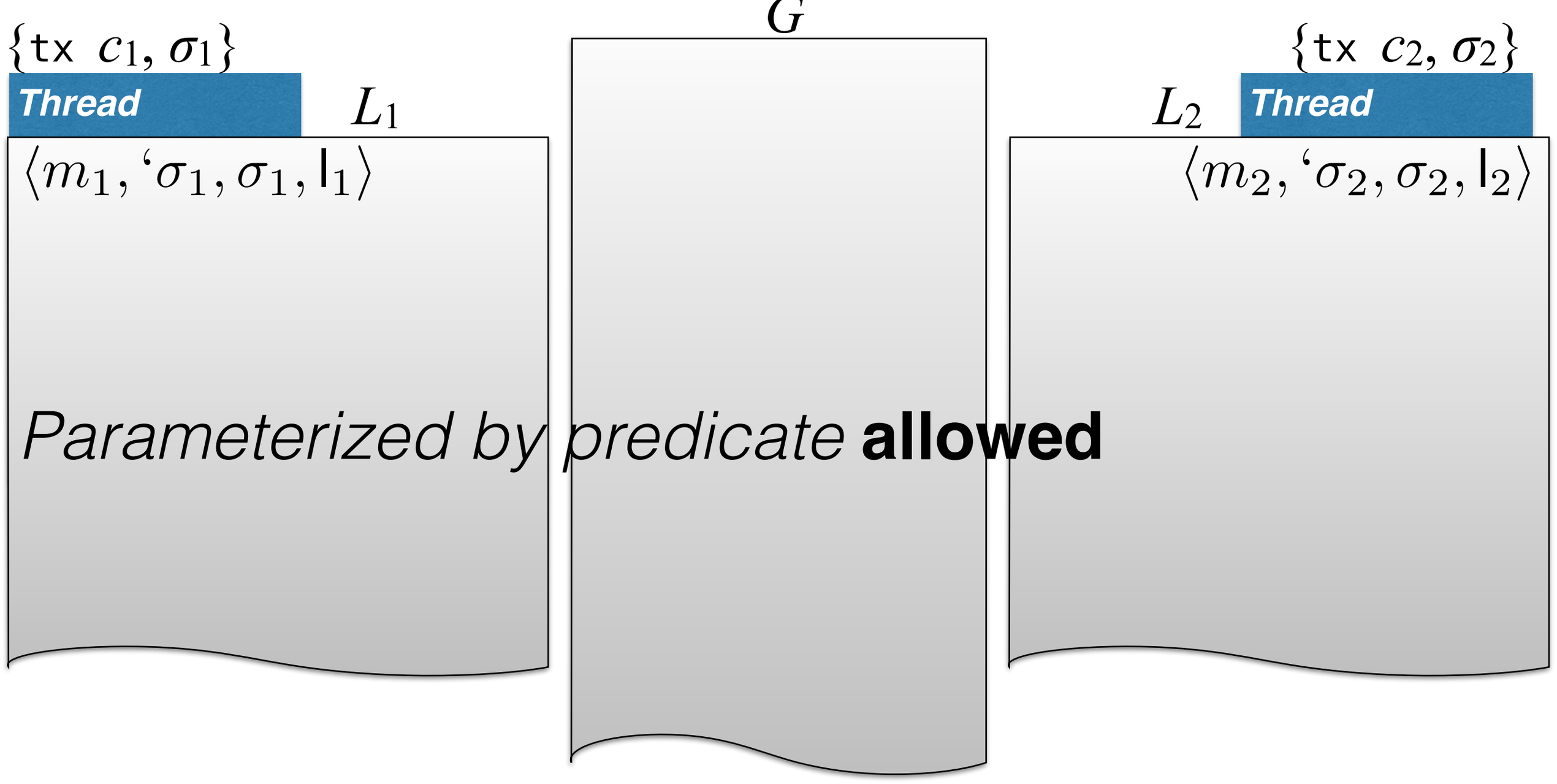
Two kinds of logs



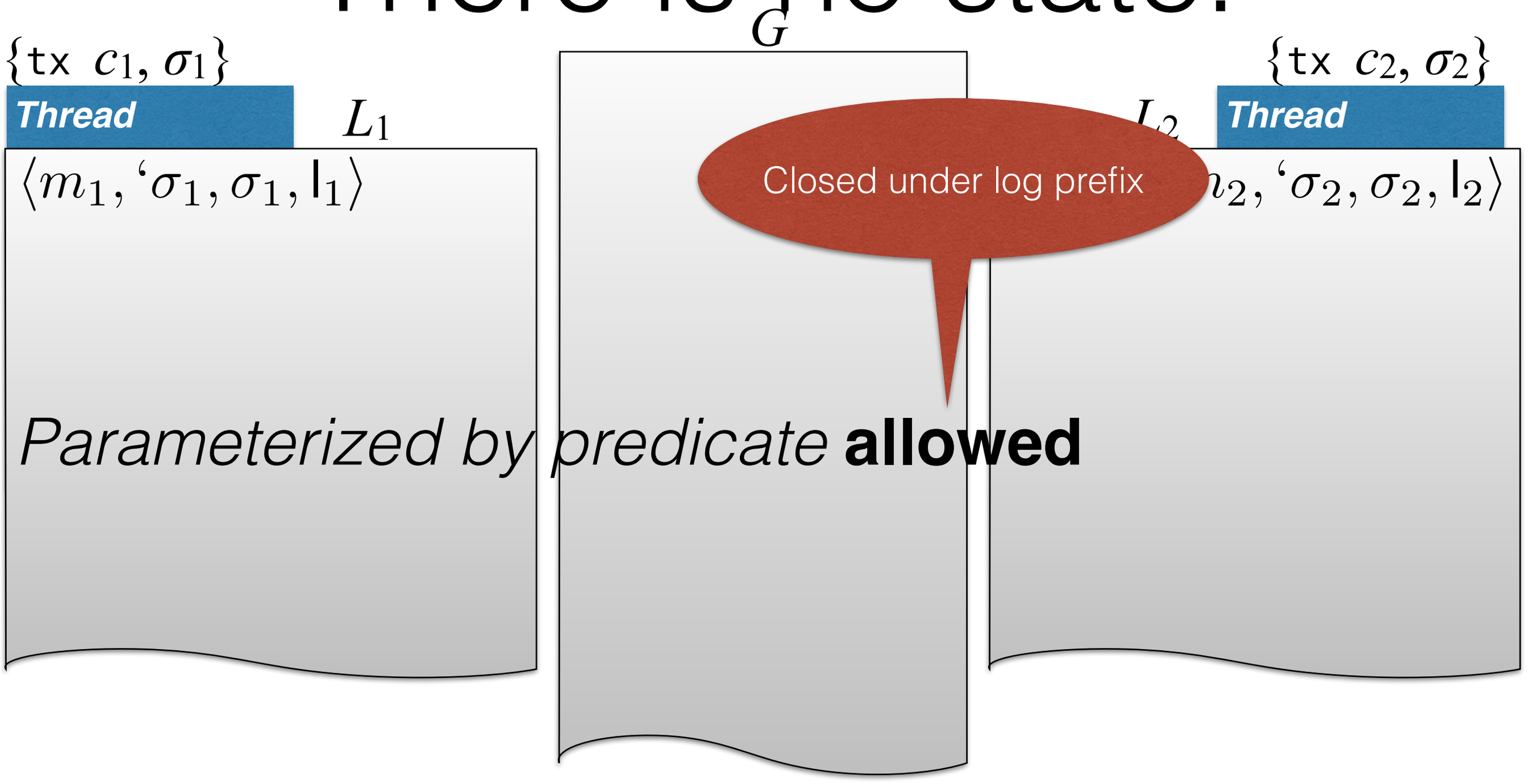
There is no state.



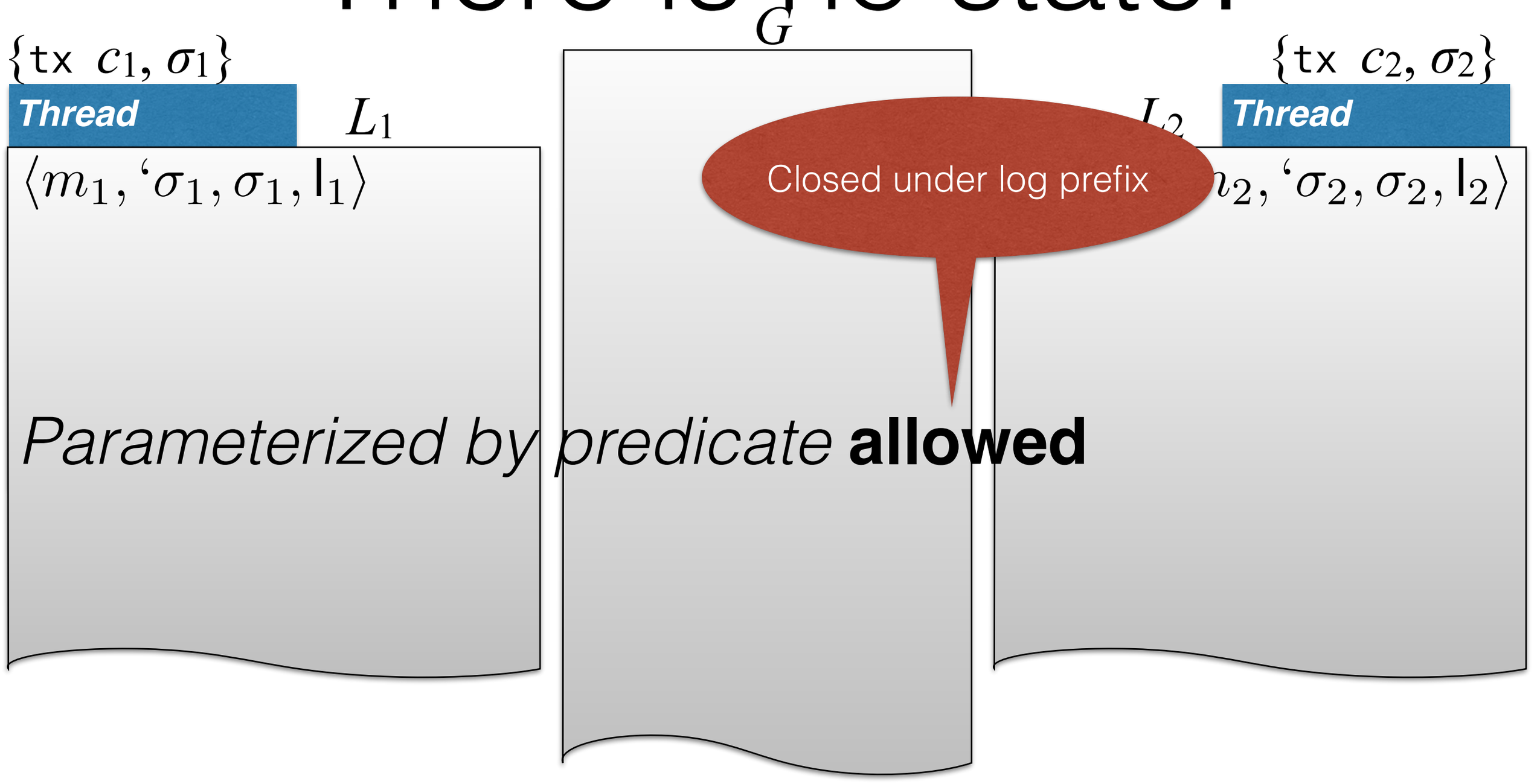
There is no state.



There is no state.

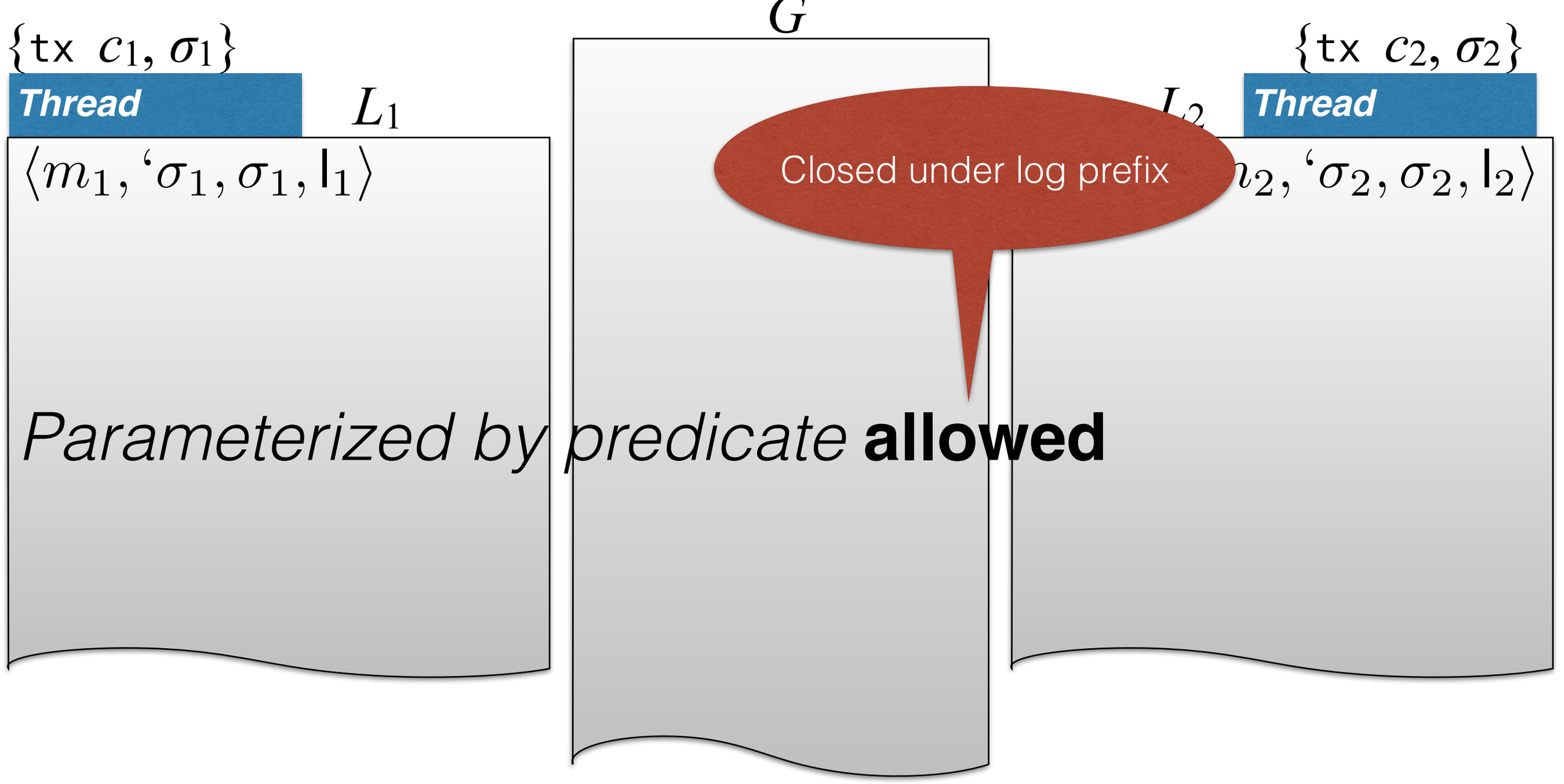


There is no state.



Log Equality

There is no state.



Log Equality

$$\frac{\mathit{allowed}\ \ell_1 \Rightarrow \mathit{allowed}\ \ell_2 \quad \forall op.\ (\ell_1 \cdot op) \preccurlyeq (\ell_2 \cdot op)}{\ell_1 \preccurlyeq \ell_2}$$

There is no state.

G

$\{\mathtt{tx}\ c_1, \sigma_1\}$

Thread

L_1

$\langle m_1, `\sigma_1, \sigma_1, \mathsf{l}_1 \rangle$

$\{\mathtt{tx}\ c_2, \sigma_2\}$

Thread

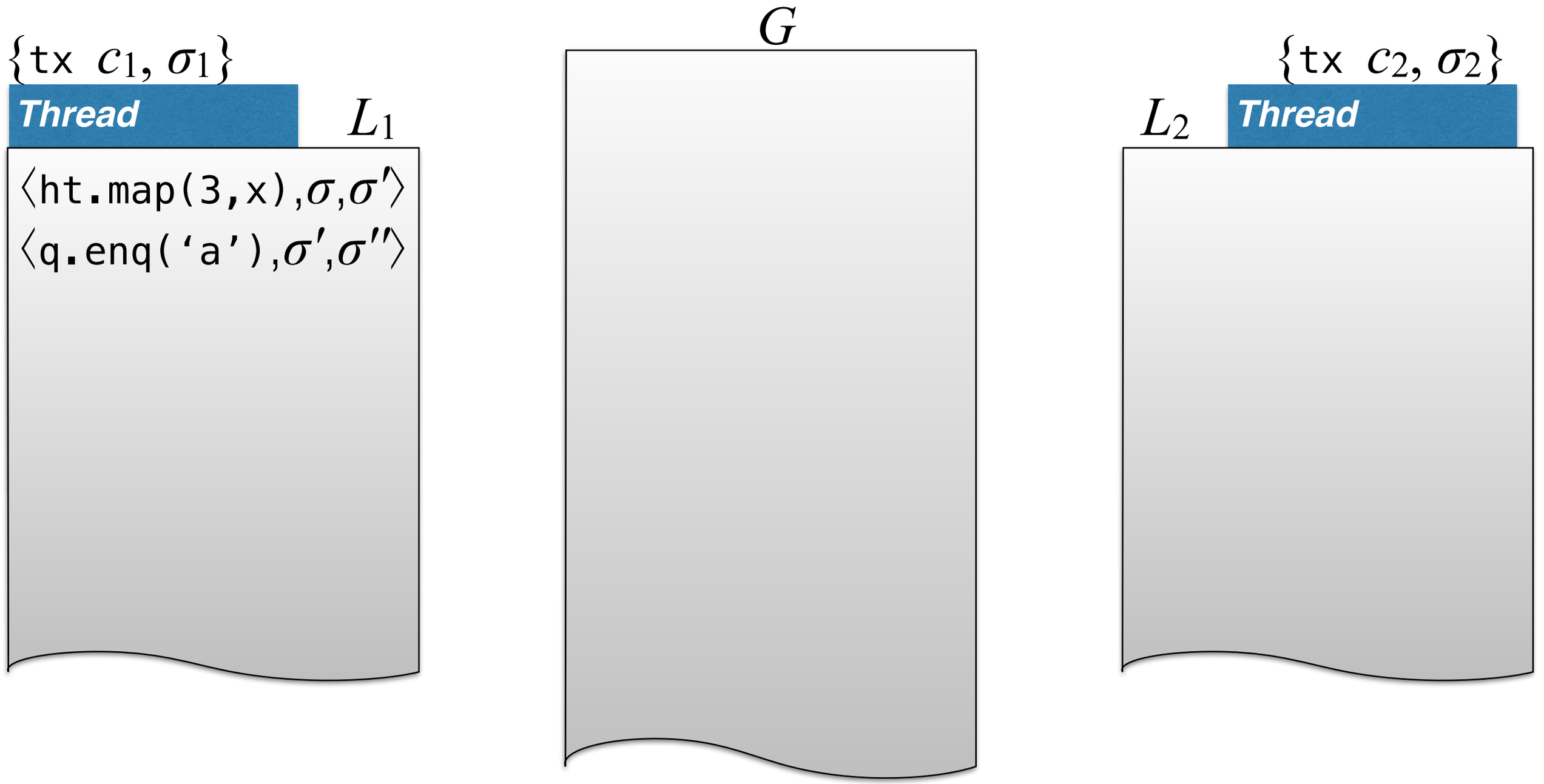
L_2

$\langle m_2, `\sigma_2, \sigma_2, \mathsf{l}_2 \rangle$

7 Simple Rules:

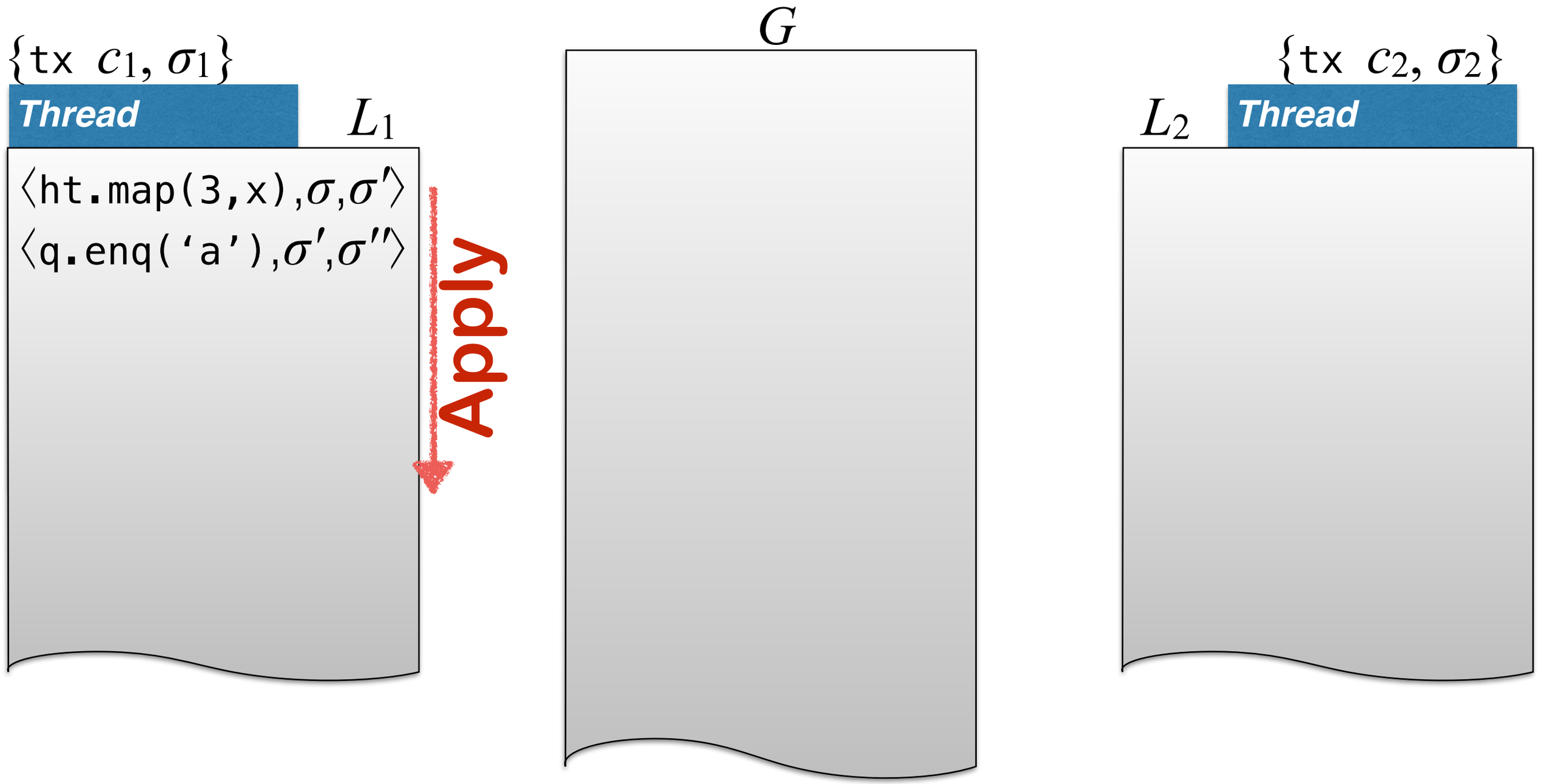
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



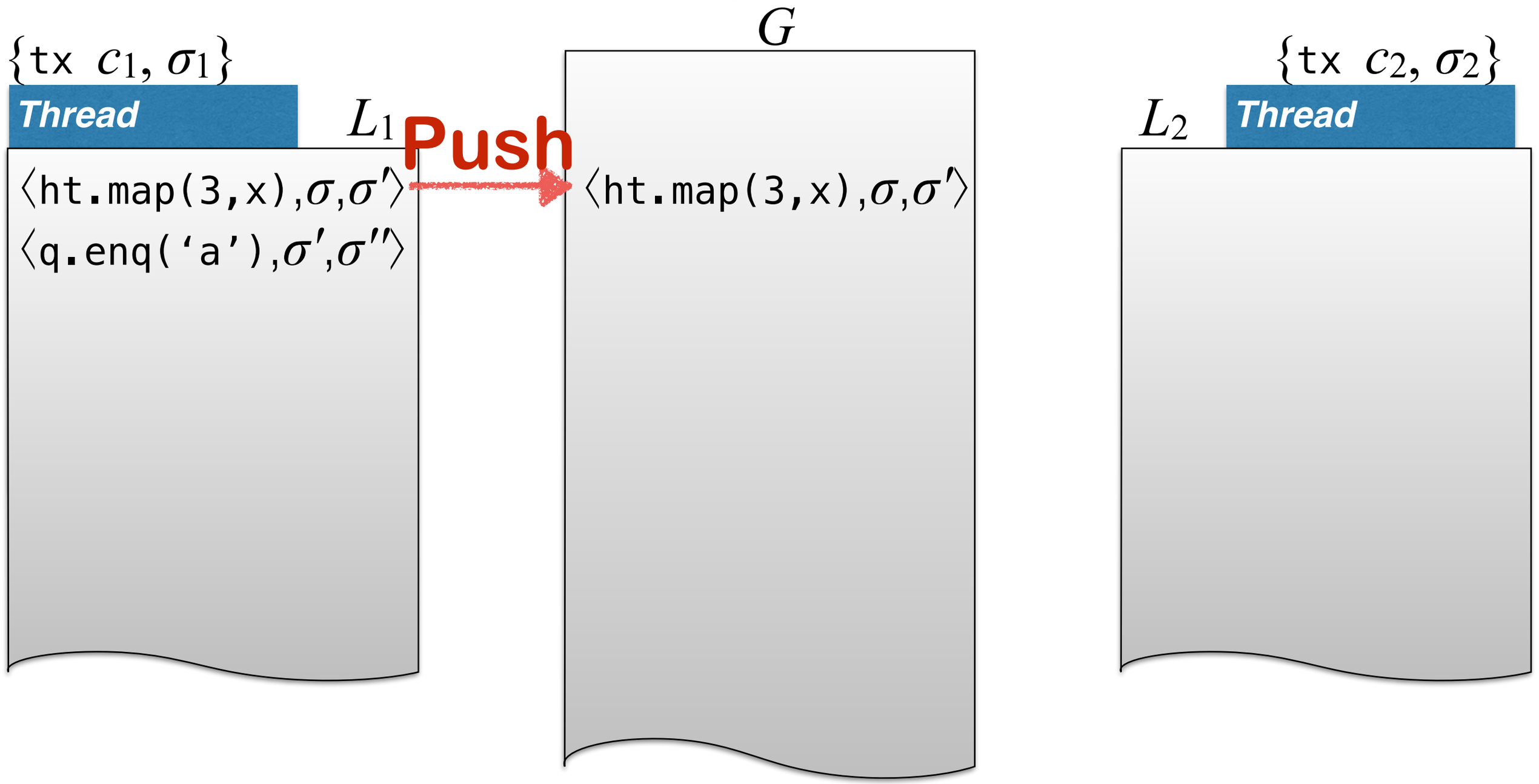
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



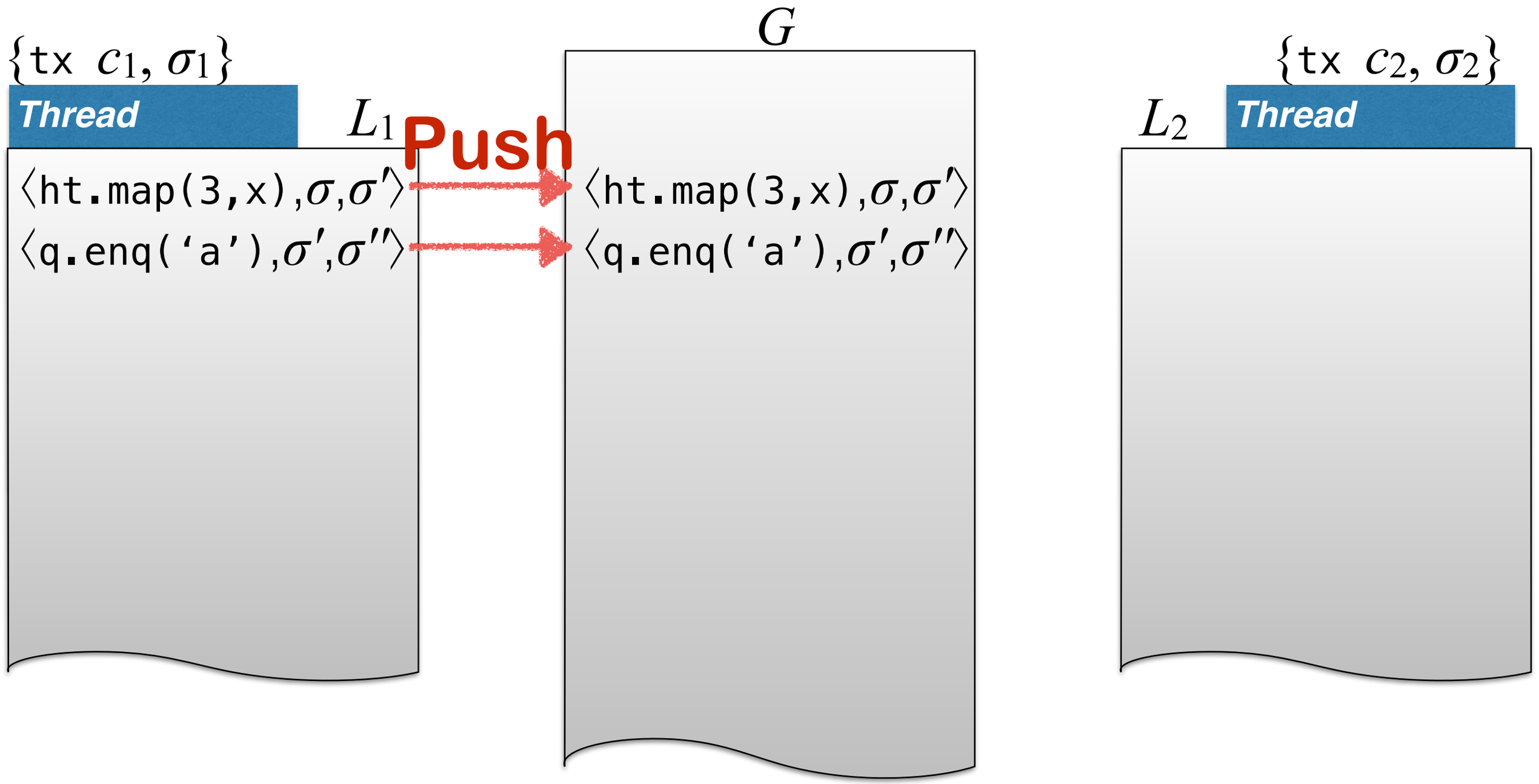
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



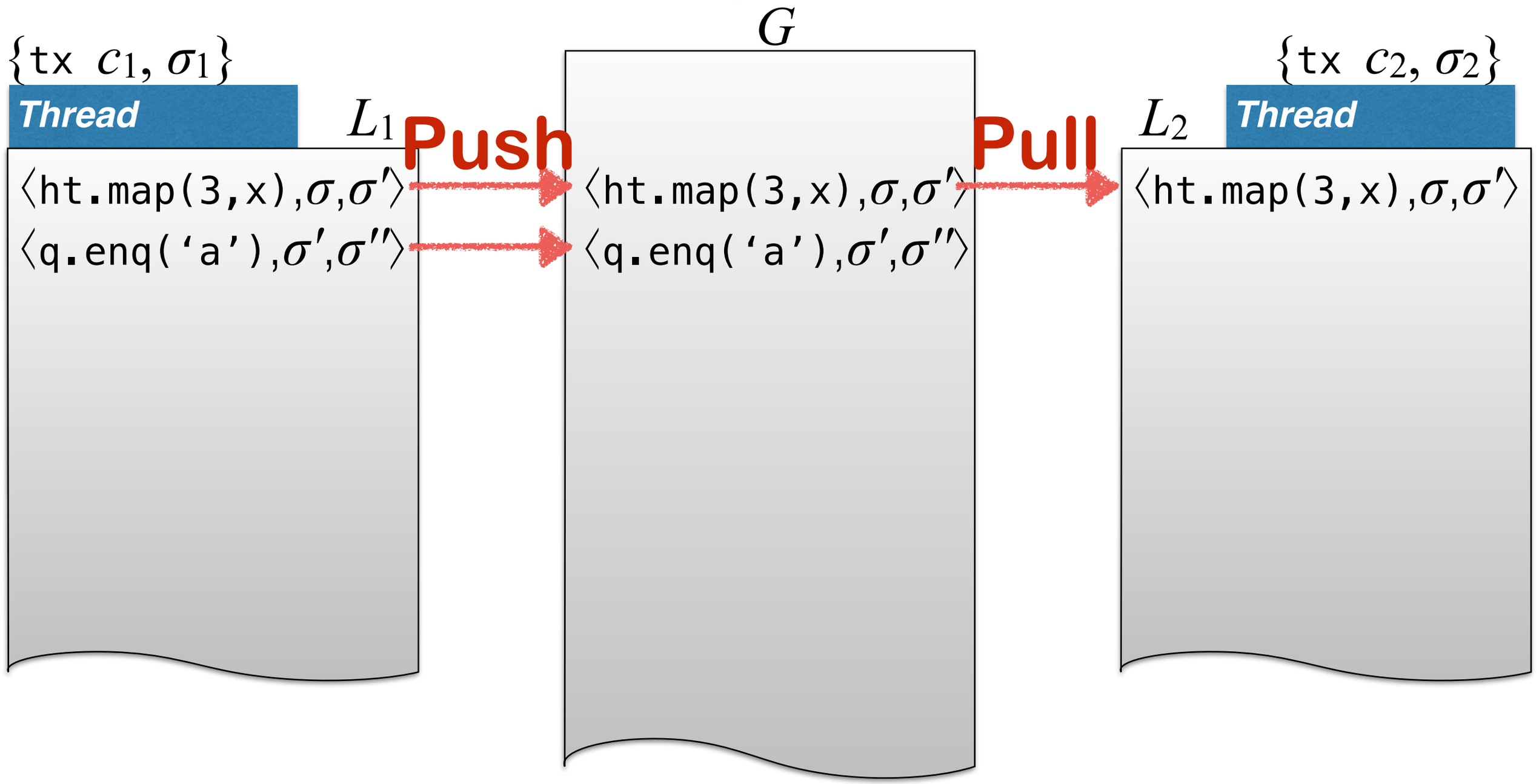
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



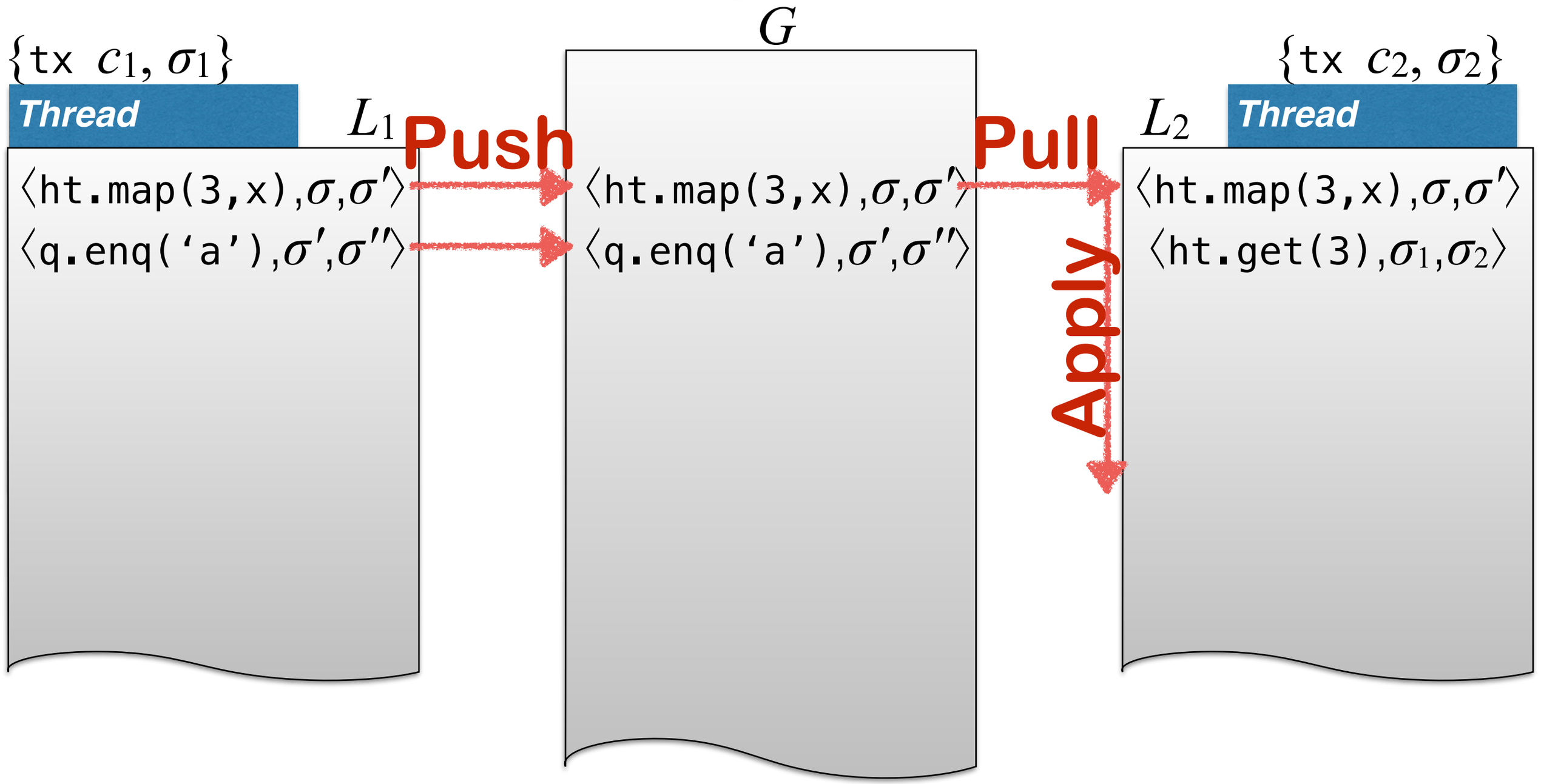
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



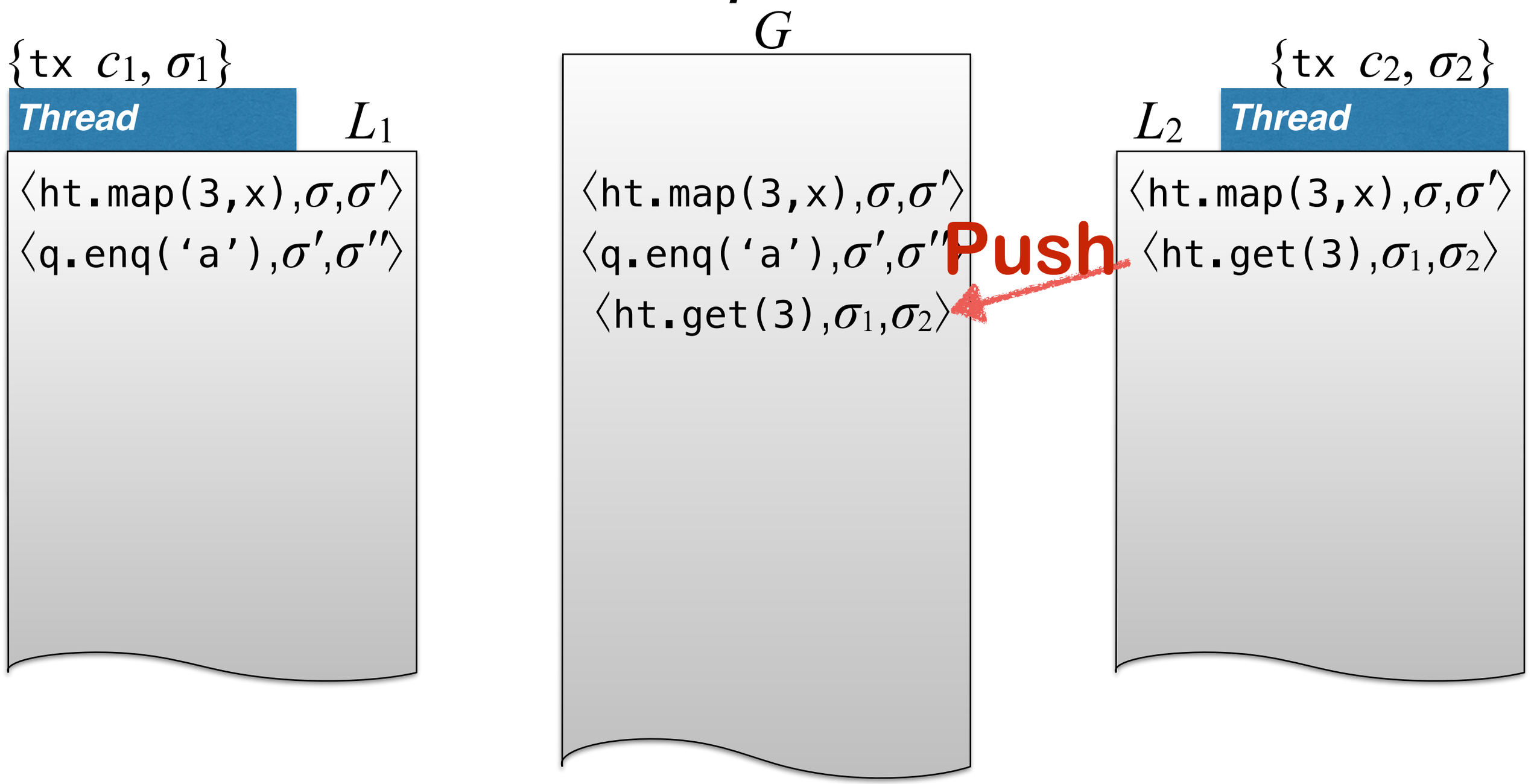
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



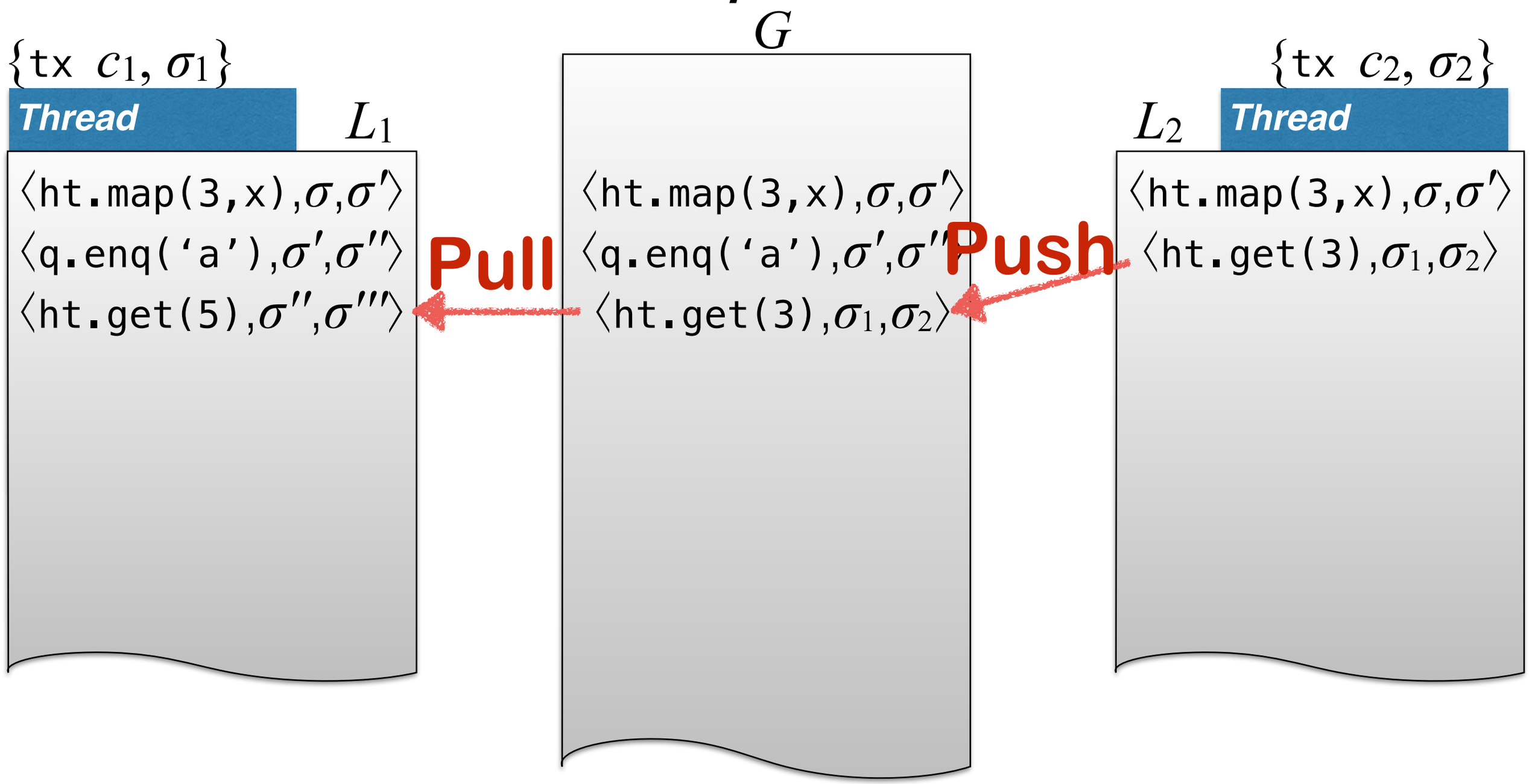
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



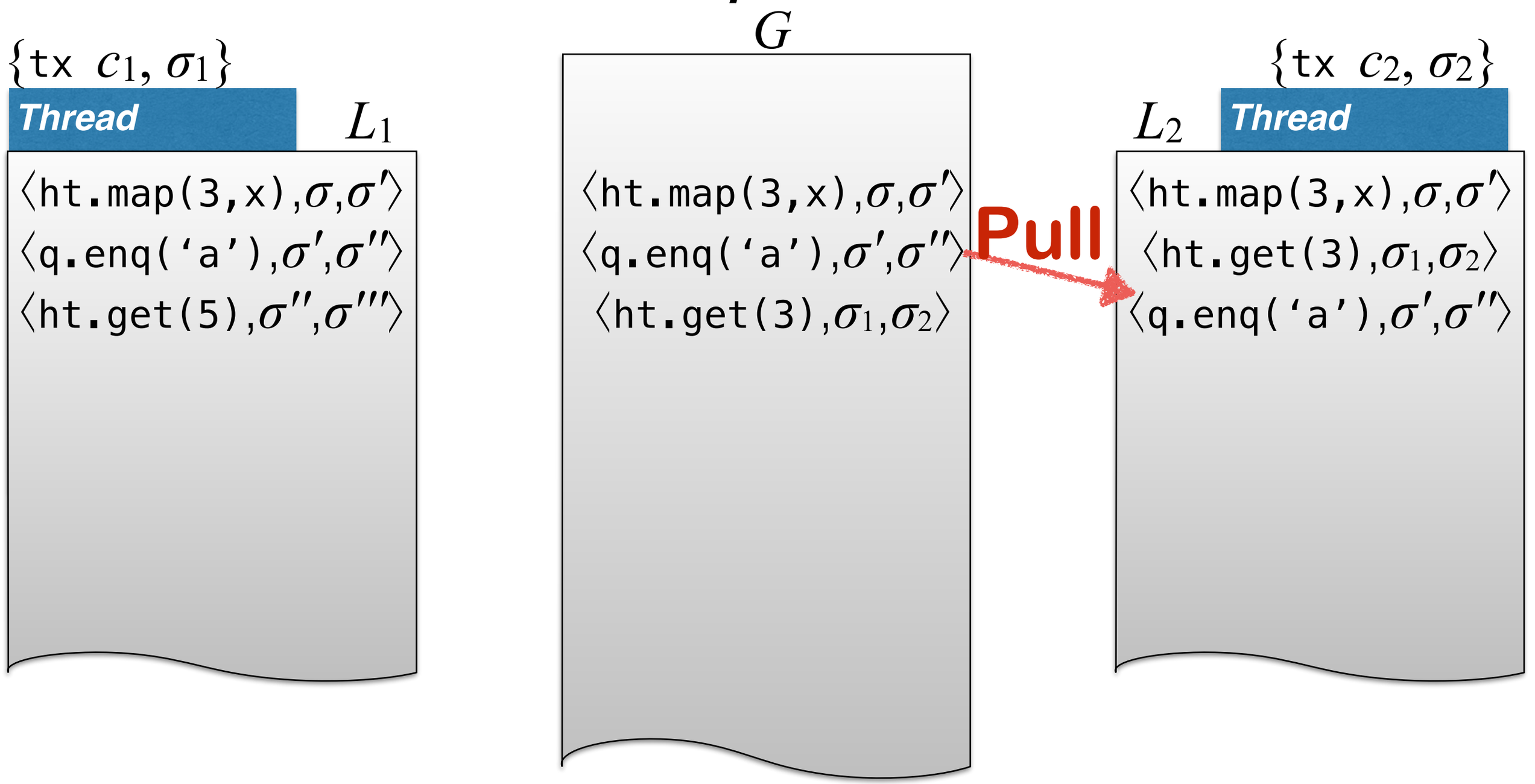
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



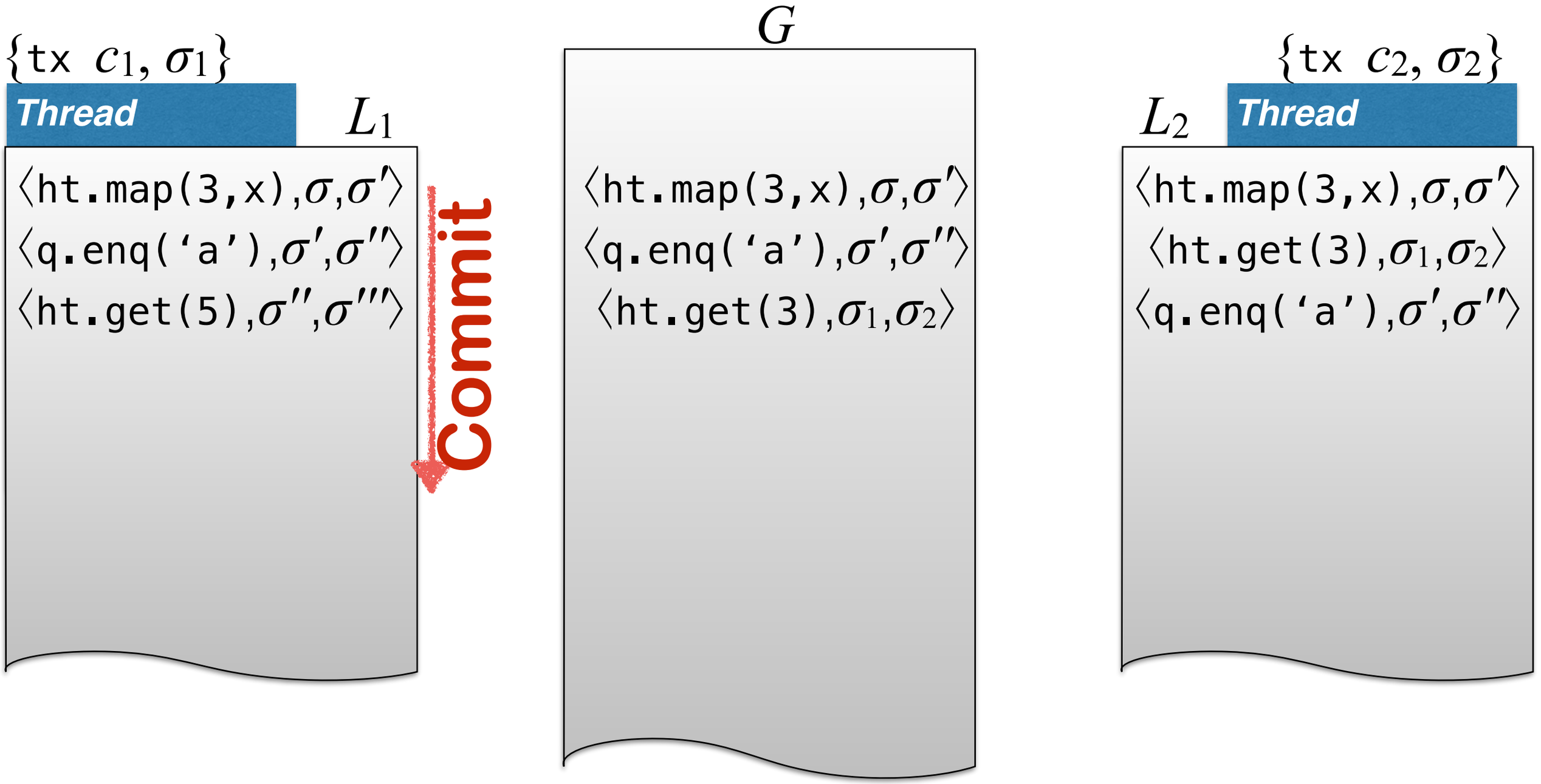
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



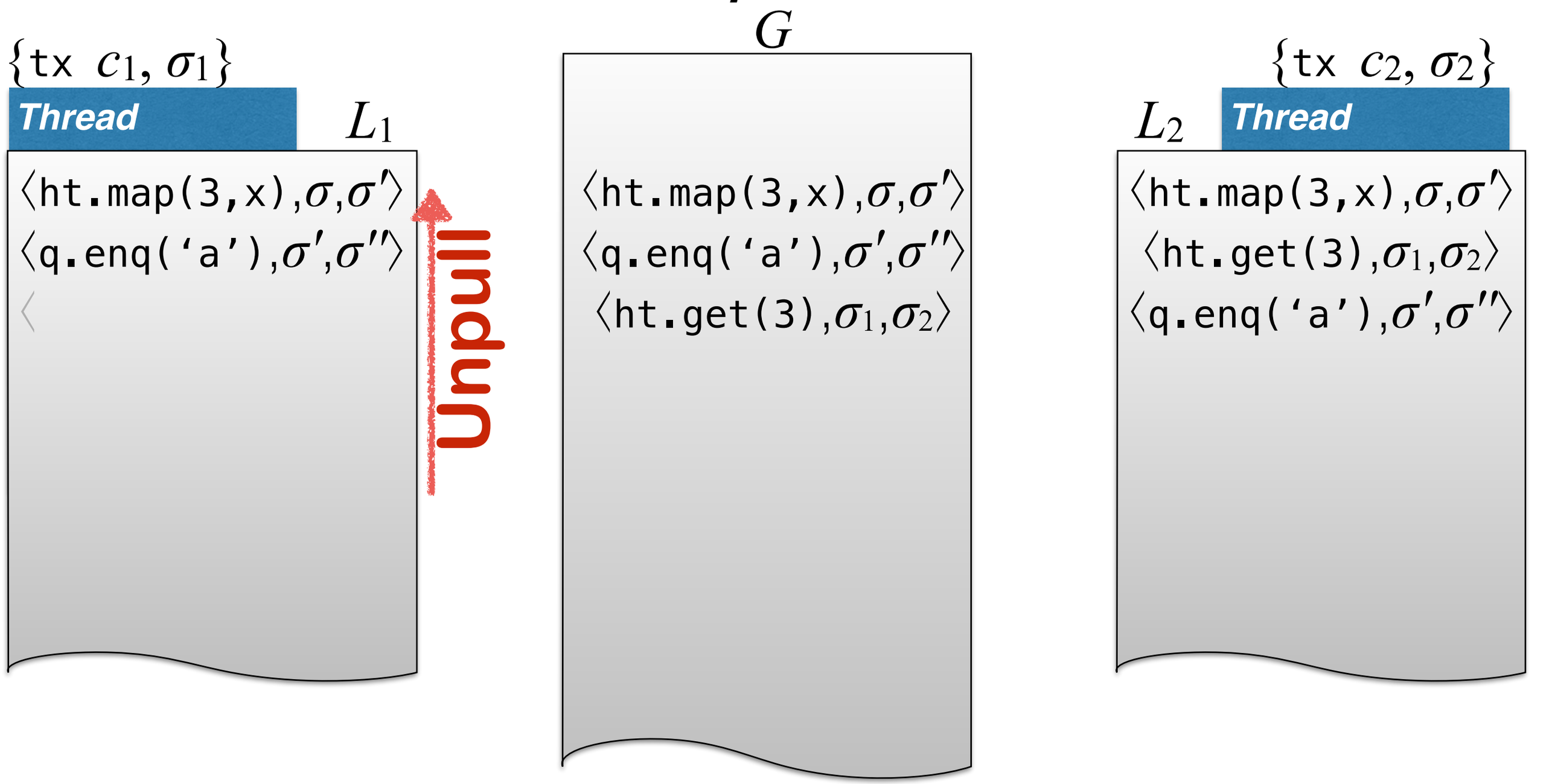
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



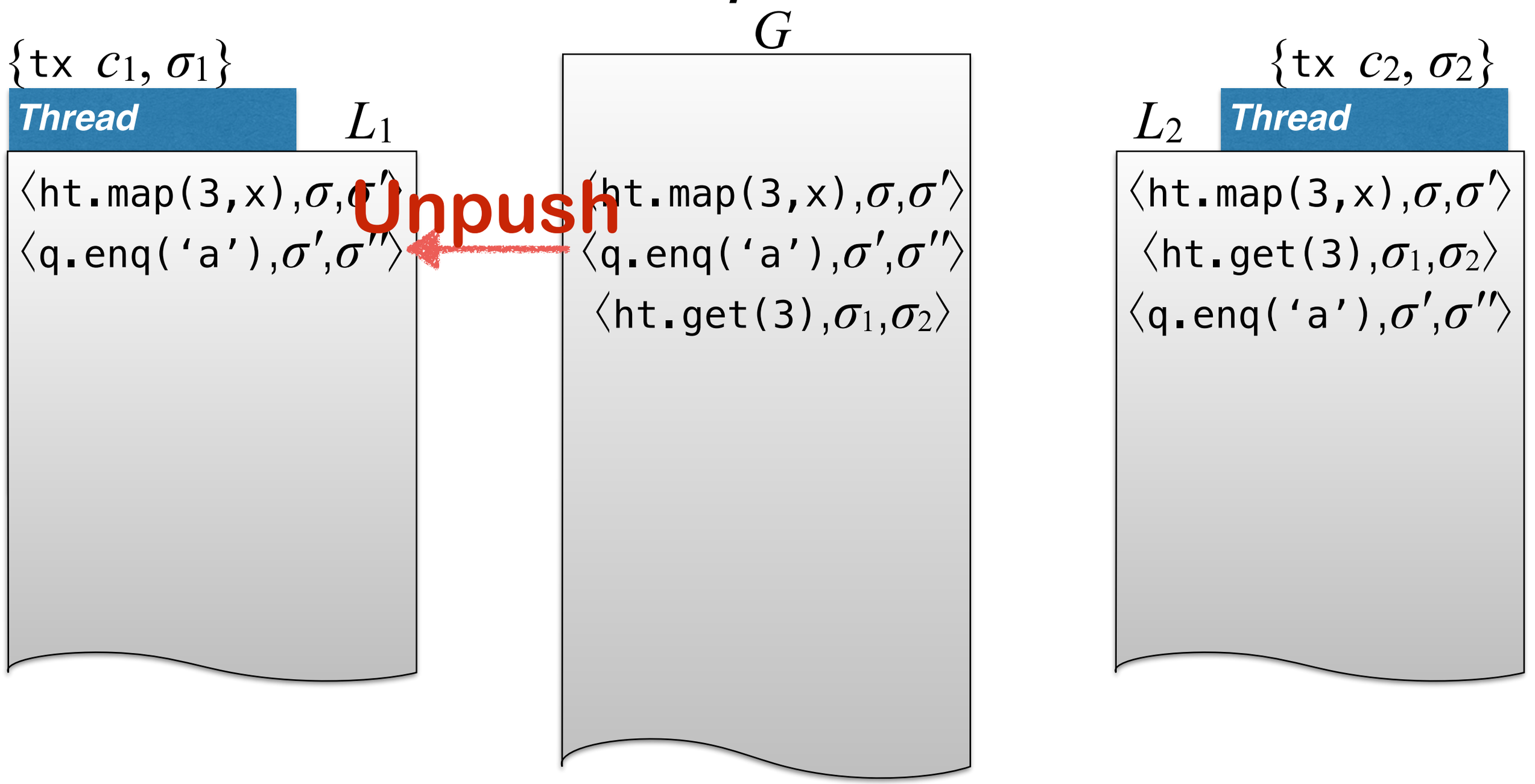
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



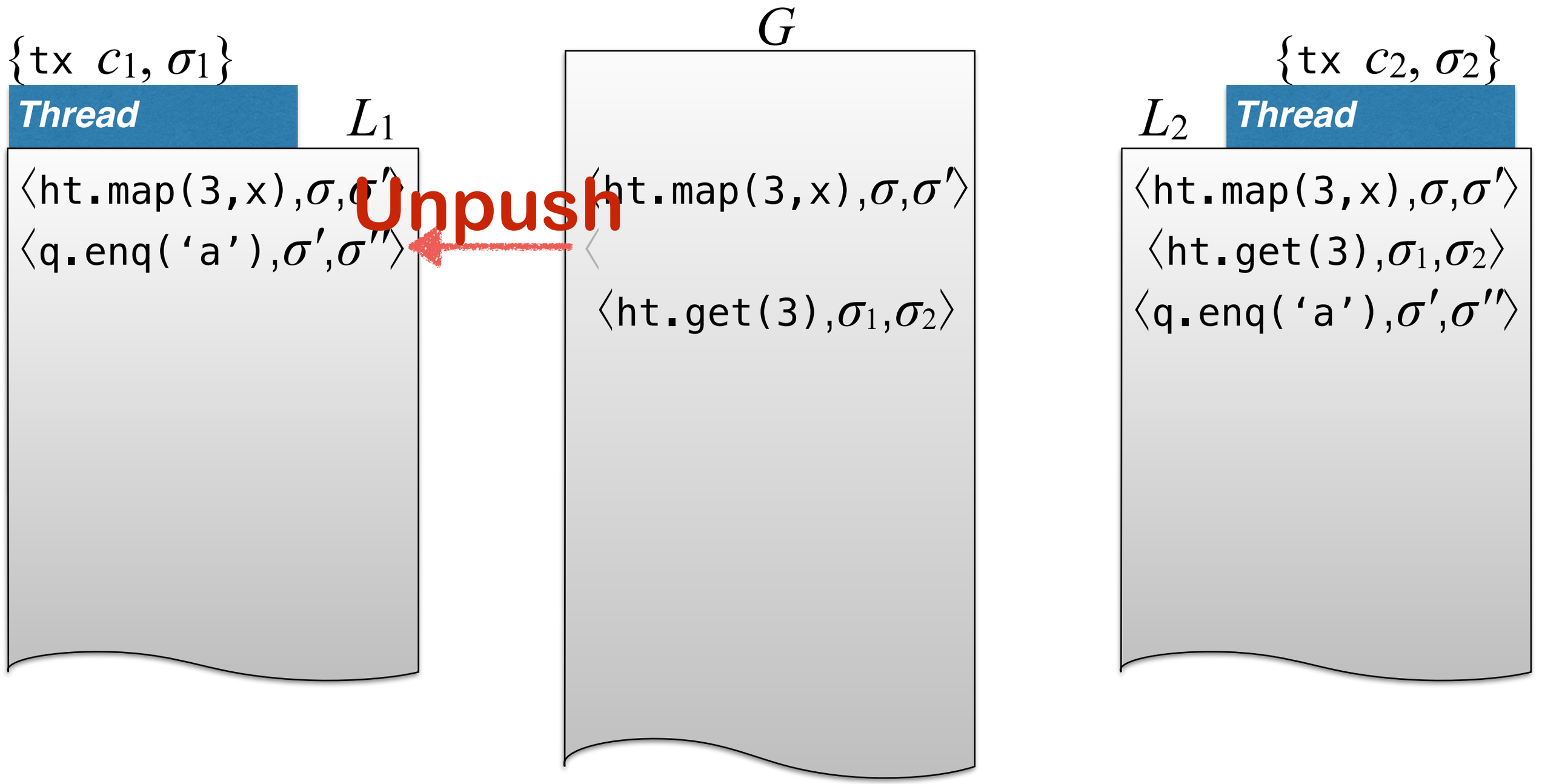
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



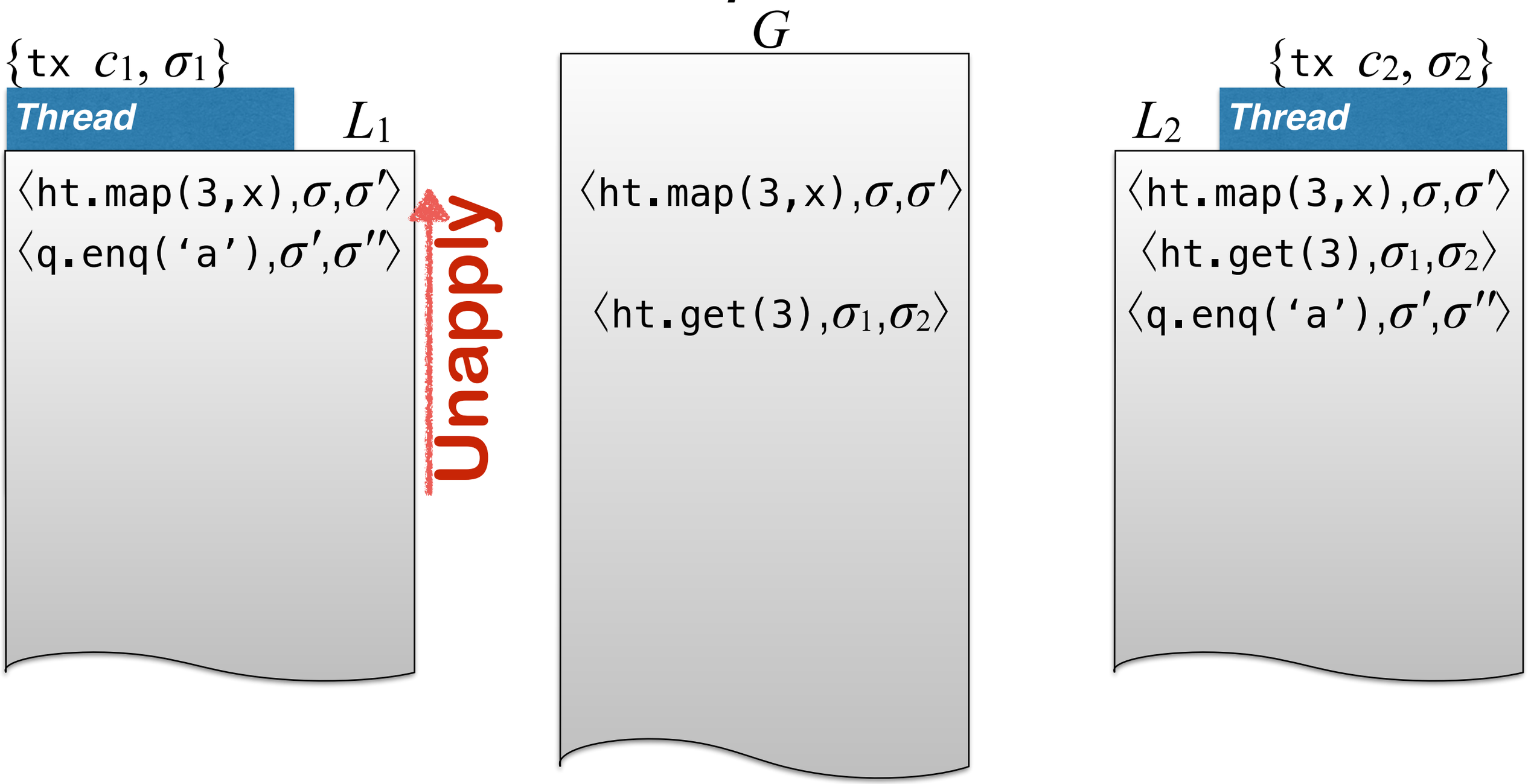
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



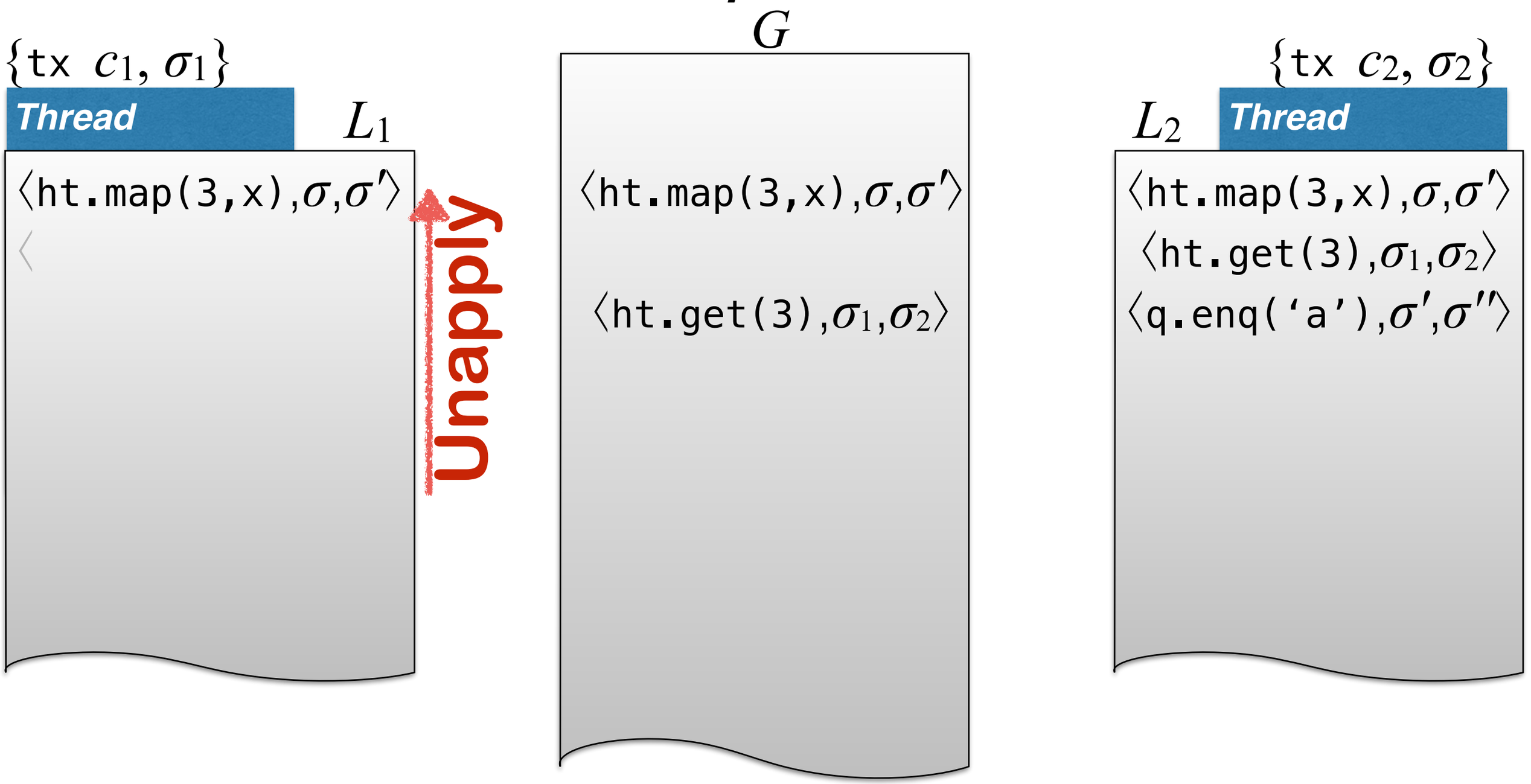
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



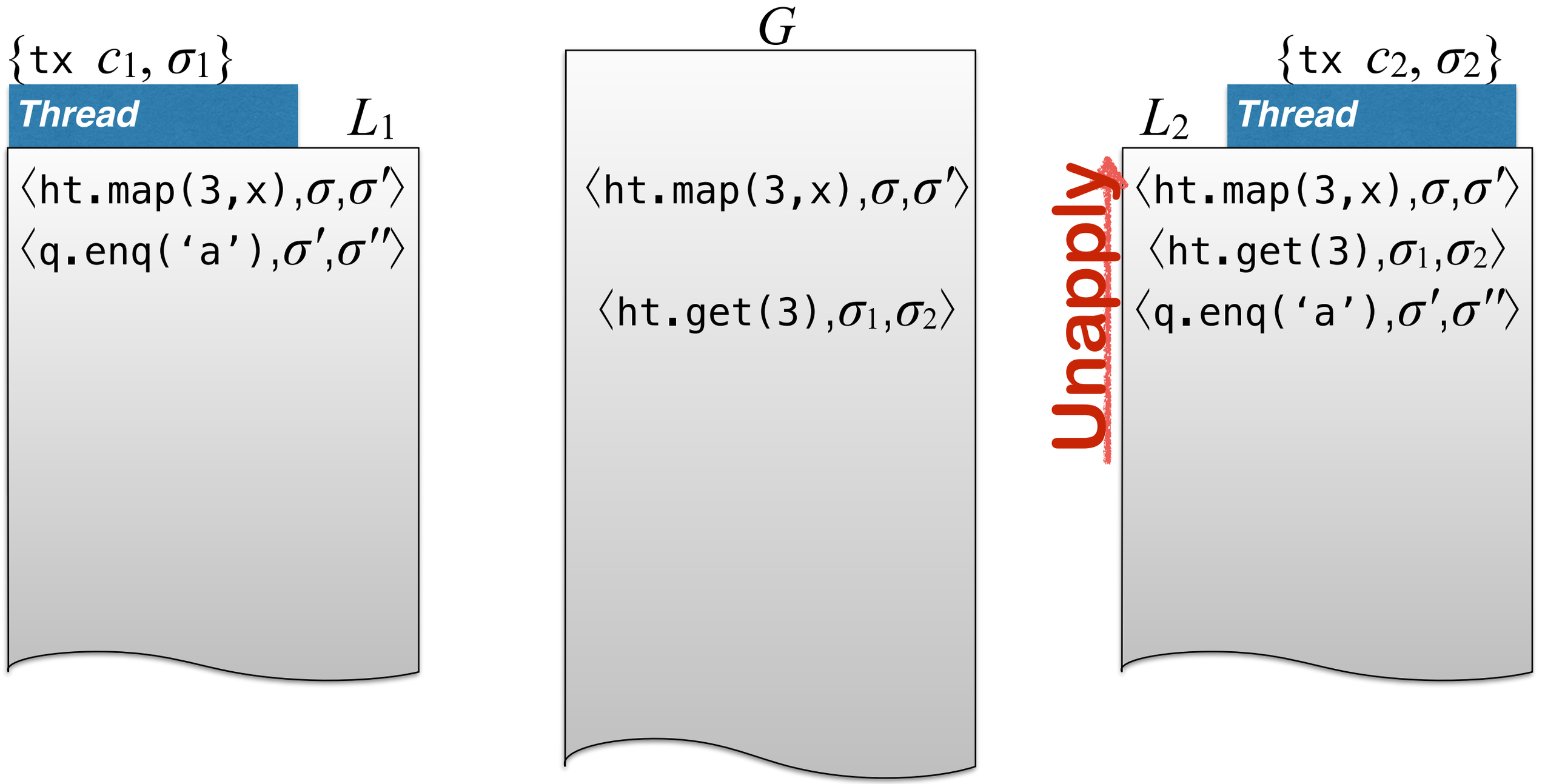
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



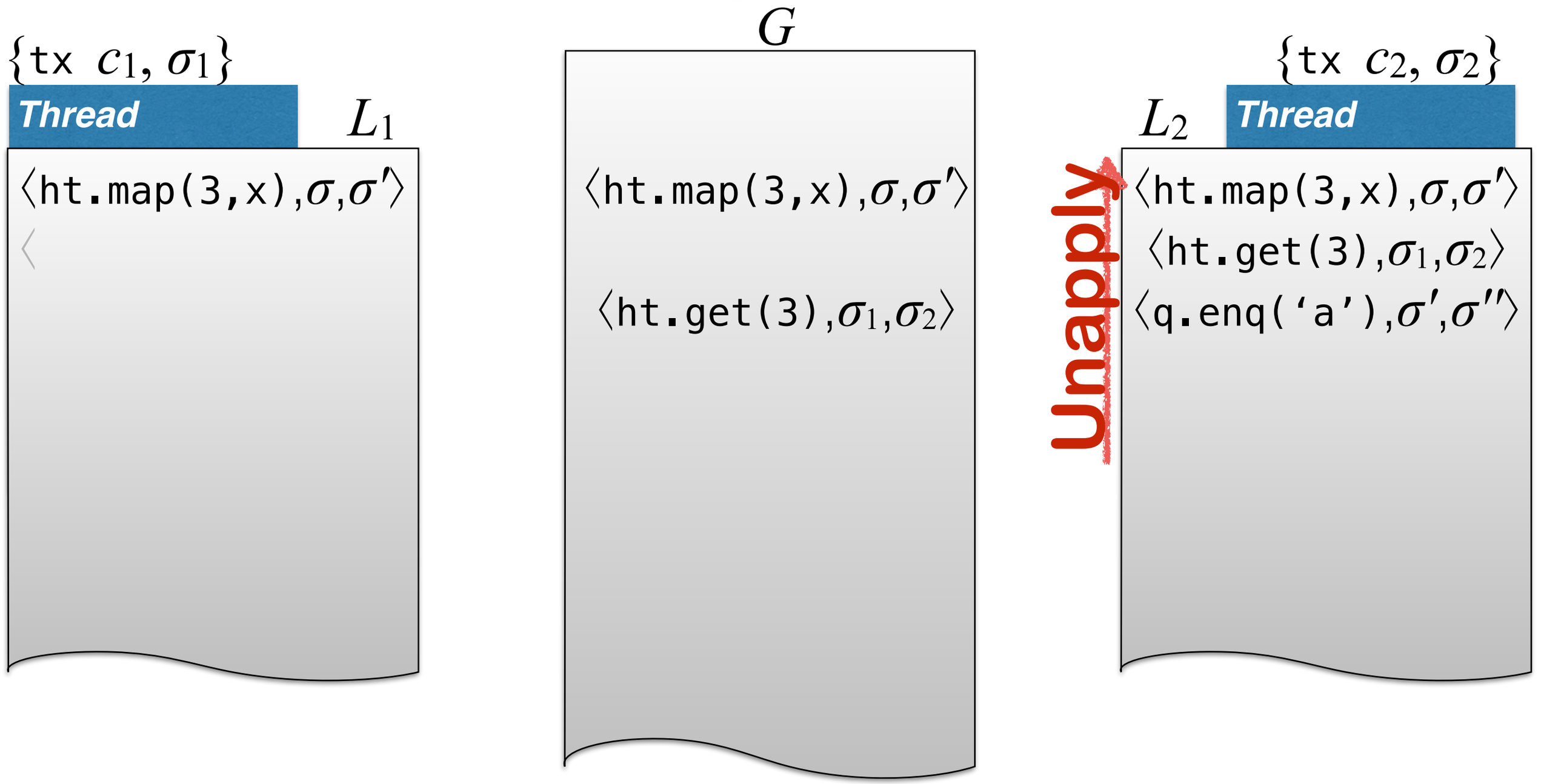
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



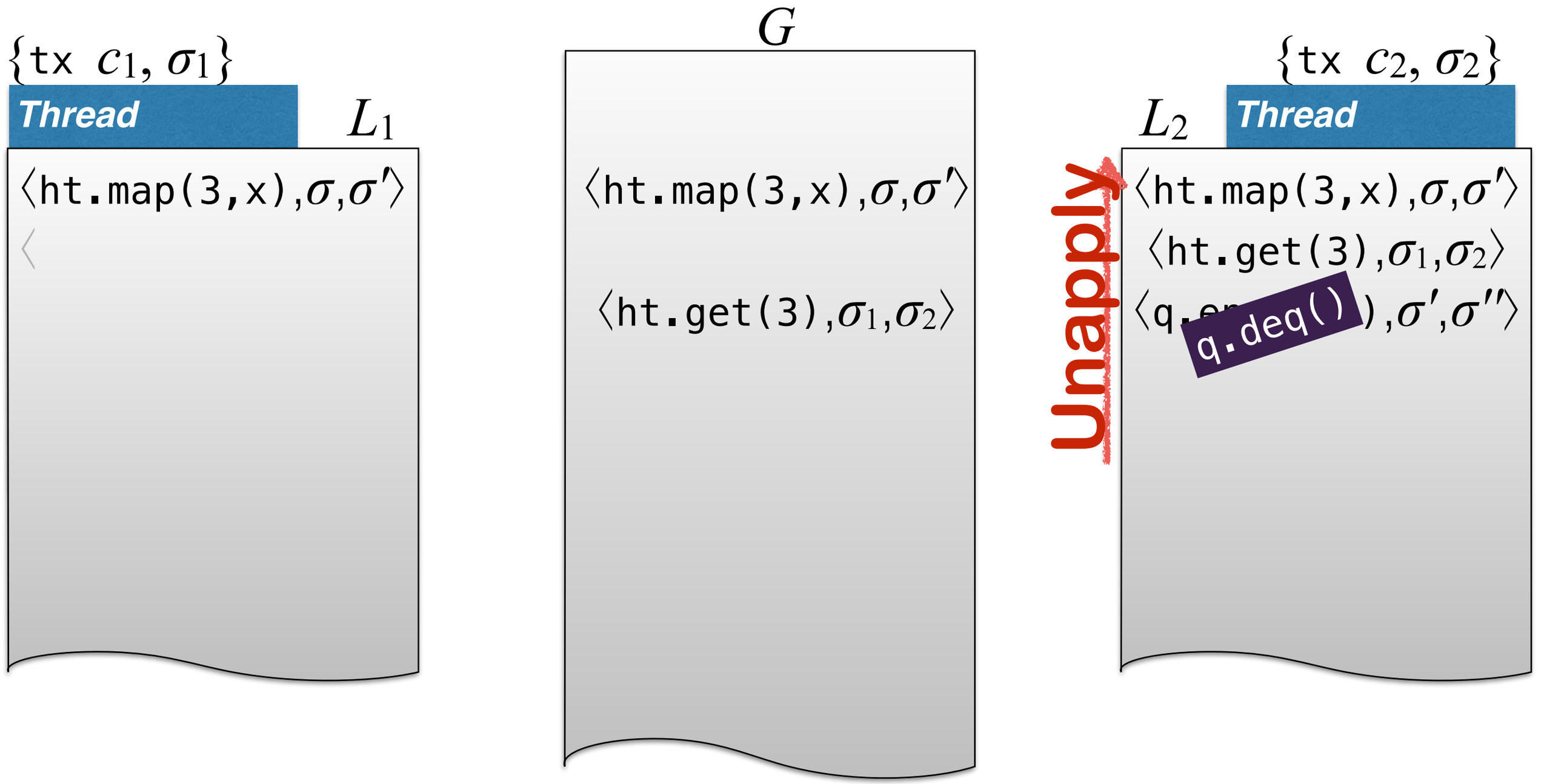
Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



Apply Push Pull Unpull Unpush Unapply Commit

The Push/Pull Model



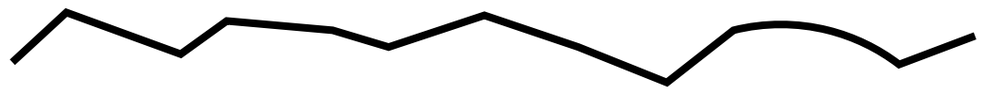
Apply Push Pull Unpull Unpush Unapply Commit

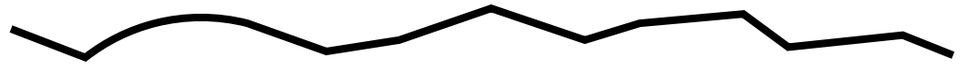
Simple (Intuitive?) Model

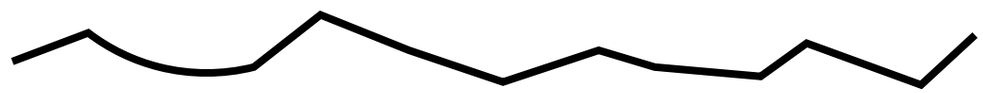
Simple (Intuitive?) Model

Getting it all to work . . .

Push/Pull Rule

Criterion (*i*): 

Criterion (*ii*): 

Criterion (*iii*): 

$$\{\texttt{tx}\ c, \sigma_1, L_1\}, G \rightarrow \{\texttt{tx}\ c', \sigma', L'\}, G'$$ APPLY

Apply Push Pull Unpull Unpush Unapply Commit

Machine Rule

$$T_1 \cdot \{\texttt{tx}\ c, \sigma_1, L_1\} \cdot T_2, G \rightarrow T_1 \cdot \{\texttt{tx}\ c', \sigma', L'\} \cdot T_2, G'$$

Push/Pull Rule

Criterion (*i*):

Criterion (*ii*):

Criterion (*iii*):

$$\frac{}{\{\texttt{tx}\ c, \sigma_1, L_1\}, G \rightarrow \{\texttt{tx}\ c', \sigma', L'\}, G'}\ \textsc{Apply}$$

Apply Push Pull Unpull Unpush Unapply Commit

Thread

$\langle$`ht.map(3,x)`,σ,$\sigma'\rangle$
$\langle$`q.enq('a')`,σ',$\sigma''\rangle$

Apply

Criterion (i):

Criterion (ii):

Criterion (iii):

$$\frac{}{\{\texttt{tx}\ c, \sigma_1, L_1\}, G \rightarrow \{\texttt{tx}\ c', \sigma_2, L_1 \cdot [\langle m, \sigma_1, \sigma_2, id\rangle, \text{unpushed } c]\}, G}\ \textsc{Apply}$$

Apply **Push Pull Unpull Unpush Unapply Commit**

Thread

$\langle$`ht.map(3,x)`,σ,$\sigma'\rangle$
$\langle$`q.enq('a')`,σ',$\sigma''\rangle$

Apply

Criterion (i): $(m_1, c_2) \in \textbf{step}(c)$

Criterion (ii):

Criterion (iii):

$$\frac{}{\{\texttt{tx}\ c, \sigma_1, L_1\}, G \rightarrow \{\texttt{tx}\ c_2, \sigma_2, L_1 \cdot [\langle m, \sigma_1, \sigma_2, id\rangle, \text{unpushed } c]\}, G}\ \textsc{Apply}$$

Apply **Push Pull Unpull Unpush Unapply Commit**

Thread

$\langle$`ht.map(3,x)`,σ,$\sigma'\rangle$
$\langle$`q.enq('a')`,σ',$\sigma''\rangle$

Apply

step(c)**:** The pair $(m, c') \in$ **step**(c) if m is a next reachable method call in the reduction of c, with remaining code c'.

fin(c)**:** This predicate is true provided that there is a reduction of c to `skip` that does not encounter a method call.

Criterion (i): $(m_1, c_2) \in$ **step**(c)

Criterion (ii):

Criterion (iii):

$$\frac{}{\{\texttt{tx}\ c, \sigma_1, L_1\}, G \rightarrow \{\texttt{tx}\ c_2, \sigma_2, L_1 \cdot [\langle m, \sigma_1, \sigma_2, id\rangle, \text{unpushed } c]\}, G}\ \textsc{Apply}$$

Apply **Push Pull Unpull Unpush Unapply Commit**

Thread

$\langle$`ht.map(3,x)`,σ,$\sigma'\rangle$
$\langle$`q.enq('a')`,σ',$\sigma''\rangle$

Apply

$$c \quad ::= \quad c_1 \ + \ c_2 \mid c_1 \ ; \ c_2 \mid (c)^* \mid \textit{skip} \mid \textit{tx}\, c \mid m$$

step(c)**:** The pair $(m, c') \in \mathsf{step}(c)$ if m is a next reachable method call in the reduction of c, with remaining code c'.

fin(c)**:** This predicate is true provided that there is a reduction of c to `skip` that does not encounter a method call.

Criterion (i): $(m_1, c_2) \in \mathbf{step}(c)$

Criterion (ii):

Criterion (iii):

$$\frac{}{\{\texttt{tx}\ c, \sigma_1, L_1\}, G \rightarrow \{\texttt{tx}\ c_2, \sigma_2, L_1 \cdot [\langle m, \sigma_1, \sigma_2, id\rangle, \mathsf{unpushed}\ c]\}, G} \textsc{Apply}$$

Apply **Push Pull Unpull Unpush Unapply Commit**

Thread

⟨ht.map(3,x),σ,σ'⟩
⟨q.enq('a'),σ',σ''⟩

Apply

$$c \quad ::= \quad c_1 \ + \ c_2 \mid c_1 \ ; \ c_2 \mid (c)^* \mid \textit{skip} \mid \textit{tx}\, c \mid m$$

step(c)**:** The pair $(m, c') \in \mathsf{step}(c)$ if m is a next reachable method call in the reduction of c, with remaining code c'.

fin(c)**:** This predicate is true provided that there is a reduction of c to `skip` that does not encounter a method call.

Criterion (i): $(m_1, c_2) \in \mathbf{step}(c)$

Criterion (ii): L_1 allows $\langle m,\sigma_1,\sigma_2,id\rangle$

Criterion (iii):

$$\frac{}{\{\mathtt{tx}\ c, \sigma_1, L_1\},G \rightarrow \{\mathtt{tx}\ c_2, \sigma_2, L_1{\cdot}[\langle m,\sigma_1,\sigma_2,id\rangle,\mathsf{unpushed}\ c]\},G}\ \textsc{Apply}$$

Apply **Push Pull Unpull Unpush Unapply Commit**

Thread

$\langle$`ht.map(3,x)`,σ,$\sigma'\rangle$
$\langle$`q.enq('a')`,σ',$\sigma''\rangle$

Apply

$$c \quad ::= \quad c_1 \ + \ c_2 \mid c_1 \ ; \ c_2 \mid (c)^* \mid \textit{skip} \mid \textit{tx}\, c \mid m$$

step(c)**:** The pair $(m, c') \in \mathsf{step}(c)$ if m is a next reachable method call in the reduction of c, with remaining code c'.

fin(c)**:** This predicate is true provided that there is a reduction of c to `skip` that does not encounter a method call.

Criterion (i): $(m_1, c_2) \in \mathbf{step}(c)$

Criterion (ii): L_1 `allows` $\langle m,\sigma_1,\sigma_2,id\rangle$

Criterion (iii): `fresh`(id)

$$\frac{}{\{\texttt{tx}\ c, \sigma_1, L_1\},G \rightarrow \{\texttt{tx}\ c_2, \sigma_2, L_1{\cdot}[\langle m,\sigma_1,\sigma_2,id\rangle,\mathsf{unpushed}\ c]\},G}\ \textsc{Apply}$$

Apply **Push Pull Unpull Unpush Unapply Commit**

Thread

$\langle$`ht.map(3,x)`,σ,$\sigma'\rangle$
$\langle$`q.enq('a')`,σ',$\sigma''\rangle$

Apply

$$c \quad ::= \quad c_1 \ + \ c_2 \mid c_1 \ ; \ c_2 \mid (c)^* \mid \textit{skip} \mid \textit{tx}\, c \mid m$$

step(c)**:** The pair $(m, c') \in \mathsf{step}(c)$ if m is a next reachable method call in the reduction of c, with remaining code c'.

fin(c)**:** This predicate is true provided that there is a reduction of c to `skip` that does not encounter a method call.

Criterion (i): $(m_1, c_2) \in \mathbf{step}(c)$

Criterion (ii): L_1 `allows` $\langle m,\sigma_1,\sigma_2,id\rangle$

Criterion (iii): `fresh`(id)

$$\frac{}{\{\texttt{tx}\ c, \sigma_1, L_1\},G \rightarrow \{\texttt{tx}\ c_2, \sigma_2, L_1 \cdot [\langle m,\sigma_1,\sigma_2,id\rangle, \text{unpushed } c]\},G}\ \textsc{Apply}$$

append

Apply **Push Pull Unpull Unpush Unapply Commit**

Thread

L_1

$\langle$ht.map(3,x),σ,$\sigma'\rangle$,unpushed $'c$

$\langle$q.enq('a'),σ',$\sigma''\rangle$,pushed $'c_1$

Unapply

$$\frac{}{\{\texttt{tx}\ c', \sigma_2, L_1 \cdot [\langle m, \sigma_1, \sigma_2, id\rangle, \texttt{unpushed}\ c]\}, G \rightarrow \{\texttt{tx}\ c, \sigma_1, L_1\}, G}\ \textsc{Unapp}$$

Apply Push Pull Unpull Unpush Unapply Commit

Thread

L_1

$\langle$ht.map(3,x),σ,$\sigma'\rangle$,unpushed $'c$

$\langle$q.enq('a'),σ',$\sigma''\rangle$,pushed $'c_1$

Push

$\langle$q.enq('a'),σ',$\sigma''\rangle$,gUC

PUSH

$$\{\texttt{tx}\ c, \sigma, L_1 \cdot [op, \texttt{unpushed}\ c] \cdot L_2 \cdot\},\ G$$

$$\rightarrow$$

$$\{\texttt{tx}\ c, \sigma, L_1 \cdot [op, \texttt{pushed}\ c] \cdot L_2 \cdot\},\ G \cdot [op, \texttt{gUC}]$$

Uncommitted

Apply Push Pull Unpull Unpush Unapply Commit

Thread

L_1

$\langle$`ht.map(3,x)`,σ,$\sigma'\rangle$,unpushed $'c$

$\langle$`q.enq('a')`,σ',$\sigma''\rangle$,pushed $'c_1$

Push

$\langle$`q.enq('a')`,σ',$\sigma''\rangle$,gUC

Criterion (i): $op \blacktriangleleft \lfloor L_1 \rfloor_{\text{unpushed}}$

PUSH

$$\{\texttt{tx}\ c, \sigma, L_1 \cdot [op, \texttt{unpushed}\ c] \cdot L_2 \cdot\}, G$$

$$\rightarrow$$

$$\{\texttt{tx}\ c, \sigma, L_1 \cdot [op, \texttt{pushed}\ c] \cdot L_2 \cdot\}, G \cdot [op, \texttt{gUC}]$$

Uncommitted

Apply Push Pull Unpull Unpush Unapply Commit

Thread

L_1

$\langle$ht.map(3,x),σ,$\sigma'\rangle$,unpushed $'c$

$\langle$q.enq('a'),σ',$\sigma''\rangle$,pushed $'c_1$

Push

$\langle$q.enq('a'),σ',$\sigma''\rangle$,gUC

Left-mover over logs

$$op_1 \blacktriangleleft op_2 \quad \equiv \quad \forall \ell.\ \ell \cdot \{op_1, op_2\} \preccurlyeq \ell \cdot \{op_2, op_1\}.$$

Criterion (i): $op \blacktriangleleft \lfloor L_1 \rfloor_{\text{unpushed}}$

PUSH

$$\{\texttt{tx}\ c, \sigma, L_1 \cdot [op, \text{unpushed } c] \cdot L_2 \cdot\}, G$$
$$\rightarrow$$
$$\{\texttt{tx}\ c, \sigma, L_1 \cdot [op, \text{pushed } c] \cdot L_2 \cdot\}, G \cdot [op, \text{gUC}]$$

Uncommitted

Apply Push Pull Unpull Unpush Unapply Commit

Thread

L_1

⟨ht.map(3,x),σ,σ'⟩,unpushed $'c$

⟨q.enq('a'),σ',σ''⟩,pushed $'c_1$

Push

⟨q.enq('a'),σ',σ''⟩,gUC

Criterion (i): $op \blacktriangleleft \lfloor L_1 \rfloor_{\text{unpushed}}$ Act as if *op* happens next

Criterion (ii): $\lfloor G \rfloor_{\text{gUC}} \setminus \lfloor L_1 \cdot L_2 \rfloor_{\text{pushed}} \blacktriangleleft op$ No conflict with other uncmted

Criterion (iii): G allows op

PUSH

$$\{\text{tx } c, \sigma, L_1 \cdot [op, \text{unpushed } c] \cdot L_2 \cdot\}, G$$
$$\rightarrow$$
$$\{\text{tx } c, \sigma, L_1 \cdot [op, \text{pushed } c] \cdot L_2 \cdot\}, G \cdot [op, \text{gUC}]$$

Application: Optimism vs. Pessimism

Apply Push Pull Unpull Unpush Unapply Commit

Thread

L_1

$\langle$`ht.map(3,x)`,σ,$\sigma'\rangle$,unpushed $'c$

$\langle$`q.enq('a')`,σ',$\sigma''\rangle$,pushed $'c_1$

Push

$\langle$`q.enq('a')`,σ',$\sigma''\rangle$,gUC

Criterion (i): $op \blacktriangleleft \lfloor L_1 \rfloor_{\text{unpushed}}$ Act as if *op* happens next

Criterion (ii): $\lfloor G \rfloor_{\text{gUC}} \setminus \lfloor L_1 \cdot L_2 \rfloor_{\text{pushed}} \blacktriangleleft op$ No conflict with other uncmted

Criterion (iii): G `allows` op

$$\frac{\{\texttt{tx}\ c, \sigma, L_1 \cdot [op, \texttt{unpushed}\ c] \cdot L_2 \cdot\},\ G \ \rightarrow\ \{\texttt{tx}\ c, \sigma, L_1 \cdot [op, \texttt{pushed}\ c] \cdot L_2 \cdot\},\ G \cdot [op, \texttt{gUC}]}{}\ \textsc{Push}$$

Application: Optimism vs. Pessimism

Apply Push Pull Unpull Unpush Unapply Commit

Thread

L_1

$\langle$`ht.map(3,x)`$,\sigma,\sigma'\rangle$,unpushed $'c$

$\langle$`q.enq('a')`$,\sigma',\sigma''\rangle$,pushed $'c_1$

Push

$\langle$`q.enq('a')`$,\sigma',\sigma''\rangle$,gUC

Criterion (i): $op \blacktriangleleft \lfloor L_1 \rfloor_{\text{unpushed}}$ — Act as if *op* happens next

Criterion (ii): $\lfloor G \rfloor_{\text{gUC}} \setminus \lfloor L_1 \cdot L_2 \rfloor_{\text{pushed}} \blacktriangleleft op$ — No conflict with other uncmted

Criterion (iii): G `allows` op

$$\frac{}{\{\texttt{tx}\ c, \sigma, L_1 \cdot [op, \texttt{unpushed}\ c] \cdot L_2 \cdot\},\ G \rightarrow \{\texttt{tx}\ c, \sigma, L_1 \cdot [op, \texttt{pushed}\ c] \cdot L_2 \cdot\},\ G \cdot [op, \texttt{gUC}]}\ \textsc{Push}$$

Application: Optimism vs. Pessimism

Apply Push Pull Unpull Unpush Unapply Commit

Thread L_1

$\langle$`ht.map(3,x)`$,\sigma,\sigma'\rangle$,`unpushed` $'c$

$\langle$`q.enq('a')`$,\sigma',\sigma''\rangle$,`pushed` $'c_1$

Pull

$\langle$`q.enq('a')`$,\sigma',\sigma''\rangle$,`gUC`

$\langle$`ht.get(3)`$,\sigma_1,\sigma_2\rangle$, g

Criterion (i): $op \notin L$ — Didn't pull already

Criterion (ii): L `allows` op — Args/RVs abide seq. spec

Criterion (iii): $op \blacktriangleleft \lfloor L \rfloor_{\text{pushed}} \cup \lfloor L \rfloor_{\text{unpushed}}$ — Can act as if *op* happened earlier

PULL

$$\{\texttt{tx}\ c, \sigma, L\}, G_1 \cdot [op, g] \cdot G_2 \rightarrow \{\texttt{tx}\ c, \sigma, L \cdot [op, \texttt{pulled}]\}, G_1 \cdot [op, g] \cdot G_2$$

Application: Opacity [GK'08] and dependent transactions [RRHW'09]

Apply Push Pull Unpull Unpush Unapply Commit

Thread

L_1

$\langle$`ht.map(3,x)`,σ,$\sigma'\rangle$,`unpushed` $'c$

$\langle$`q.enq('a')`,σ',$\sigma''\rangle$,`pushed` $'c_1$

$\langle$`ht.get(3)`,σ_1,$\sigma_2\rangle$, `pulled`

Pull

$\langle$`q.enq('a')`,σ',$\sigma''\rangle$,`gUC`

$\langle$`ht.get(3)`,σ_1,$\sigma_2\rangle$, g

Criterion (i): $op \notin L$ — Didn't pull already

Criterion (ii): L `allows` op — Args/RVs abide seq. spec

Criterion (iii): $op \blacktriangleleft \lfloor L \rfloor_{\text{pushed}} \cup \lfloor L \rfloor_{\text{unpushed}}$ — Can act as if *op* happened earlier

PULL

$$\{\texttt{tx}\ c, \sigma, L\}, G_1 \cdot [op, g] \cdot G_2 \rightarrow \{\texttt{tx}\ c, \sigma, L \cdot [op, \texttt{pulled}]\}, G_1 \cdot [op, g] \cdot G_2$$

Application: Opacity [GK'08] and dependent transactions [RRHW'09]

Apply Push Pull Unpull Unpush Unapply Commit

Thread L_1

$\langle$`ht.map(3,x)`$,\sigma,\sigma'\rangle$,`unpushed` $'c$

$\langle$`q.enq('a')`$,\sigma',\sigma''\rangle$,`pushed` $'c_1$

$\langle$`ht.get(3)`$,\sigma_1,\sigma_2\rangle$, `pulled`

Pull

$\langle$`q.enq('a')`$,\sigma',\sigma''\rangle$,`gUC`

$\langle$`ht.get(3)`$,\sigma_1,\sigma_2\rangle$, g

Criterion (i): $op \notin L$ — Didn't pull already

Criterion (ii): L `allows` op — Args/RVs abide seq. spec

Criterion (iii): op — Can act as if op happened earlier

$$\frac{}{\{\texttt{tx}\ c, \sigma, L\}, G_1 \cdot [op, g] \cdot G_2 \rightarrow \{\texttt{tx}\ c, \sigma, L \cdot [op, \texttt{pulled}]\}, G_1 \cdot [op, g] \cdot G_2}\ \textsc{Pull}$$

Application: Opacity [GK'08] and dependent transactions [RRHW'09]

Apply Push Pull Unpull Unpush Unapply Commit

Thread

$\langle$ht.map(3,x),σ,σ',unpushed c

$\langle$q.enq('a'),σ',$\sigma''\rangle$,pushed c_1

$\langle$ht.get(3),σ_1,$\sigma_2\rangle$, pulled

L_1

Commit

$\langle$q.enq('a'),σ',$\sigma''\rangle$,gUC

$\langle$ht.get(3),σ_1,$\sigma_2\rangle$, g

Criterion (i): $\mathtt{fin}(c)$

Criterion (ii): $L \subseteq G_1$ — Pushed all my stuff

Criterion (iii): $\lfloor L \rfloor_{\mathtt{pulled}} \subseteq \lfloor G_1 \rfloor_{\mathtt{gC}}$ — Pulled all committed stuff

Criterion (iv): $\mathtt{cmt}(G_1, L, G_2)$ — Swap my flags from **gUC** to **gC**

COMMIT

$$\{\mathtt{tx}\ c, \sigma, L\}, G_1 \longrightarrow \{\mathtt{skip}, \sigma, []\}, G_2$$

Apply Push Pull Unpull Unpush Unapply Commit

$$\frac{\begin{array}{ll}(i)- & c_1 \not\downarrow\downarrow (m_1, c_2) \\ (ii)- & L_1 \text{ allows } \langle m_1, \sigma_1, \sigma_2 \rangle \\ (iii)- & \mathsf{fresh}(id)\end{array}}{\{\mathtt{tx}\ c_1, \sigma_1, L_1\}, G_1 \ \underrightarrow{\mathsf{fwd}}\ \{\mathtt{tx}\ c_2, \sigma_2, L_1 \cdot [\langle m_1, \sigma_1, \sigma_2, id\rangle, \mathsf{unpushed}\ c_1]\}, G_1}\ \text{APP}$$

$$\frac{}{\{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [\langle m_1, \sigma_2, \sigma_3, id\rangle, \mathsf{unpushed}\ c_2]\}, G_1 \ \underrightarrow{\mathsf{bwd}}\ \{\mathtt{tx}\ c_2, \sigma_2, L_1\}, G_1}\ \text{UNAPP}$$

$$\frac{\begin{array}{ll}(i)- & op \blacktriangleleft \lfloor L_1 \rfloor_{\mathsf{unpushed}} \\ (ii)- & \lfloor G_1 \rfloor_{\mathsf{gUCmt}} \setminus \lfloor L_1 \cdot L_2 \rfloor_{\mathsf{pushed}} \blacktriangleleft op \\ (iii)- & G_1 \text{ allows } op\end{array}}{\{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [op, \mathsf{unpushed}\ c_2] \cdot L_2\}, G_1 \ \underrightarrow{\mathsf{fwd}}\ \{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [op, \mathsf{pushed}\ c_2] \cdot L_2\}, G_1 \cdot [op, \mathsf{gUCmt}]}\ \text{PUSH}$$

$$\frac{\begin{array}{ll}(i)- & \text{allowed } G_1 \cdot G_2 \\ (ii)- & \lfloor L_2 \rfloor_{\mathsf{pushed}} \blacktriangleleft op\end{array}}{\{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [op, \mathsf{pushed}\ c_2] \cdot L_2\}, G_1 \cdot [op, g] \cdot G_2 \ \underrightarrow{\mathsf{bwd}}\ \{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [op, \mathsf{unpushed}\ c_2] \cdot L_2\}, G_1 \cdot G_2}\ \text{UNPUSH}$$

$$\frac{\begin{array}{ll}(i)- & op \notin L \\ (ii)- & L \text{ allows } op \\ (iii)- & op \blacktriangleleft \lfloor L \rfloor_{\mathsf{pushed}} \cup \lfloor L \rfloor_{\mathsf{unpushed}}\end{array}}{\{\mathtt{tx}\ c_1, \sigma_1, L\}, G_1 \cdot [op, g] \cdot G_2 \ \underrightarrow{\mathsf{fwd}}\ \{\mathtt{tx}\ c_1, \sigma_1, L \cdot [op, \mathsf{pulled}]\}, G_1 \cdot [op, g] \cdot G_2}\ \text{PULL}$$

$$\frac{(i)- \quad \text{allowed } L_1 \cdot L_2}{\{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot [op, \mathsf{pulled}] \cdot L_2\}, G \ \underrightarrow{\mathsf{bwd}}\ \{\mathtt{tx}\ c_1, \sigma_1, L_1 \cdot L_2\}, G}\ \text{UNPULL}$$

Apply Push Pull Unpull Unpush Unapply Commit

Theorem. The Push/Pull model is serializable.

Apply Push Pull Unpull Unpush Unapply Commit

Theorem. The Push/Pull model is serializable.

Push/Pull model guarantees that transactions are executed equivalently to just each being done atomically.

Apply Push Pull Unpull Unpush Unapply Commit

Theorem. The Push/Pull model is serializable.

Push/Pull model guarantees that transactions are executed equivalently to just each being done atomically.

Proof via simulation with atomic machine.

Apply Push Pull Unpull Unpush Unapply Commit

Evaluation


Pessimism!
Consistency
Irrevocability
Opacity!
Hardware TM
Software TM
Hybrid TM
Open Nested
Serializability!
Lazy Reads
Optimism!
Memory Ops
Eager Writes
ADT Ops
(Boosting)
Dependent Txns

Conclusions

- Can encode numerous TM schemes into this
- Informal tool for thinking about TMs
- Formally provides serializability

Questions?

Apply Push Pull Unpull Unpush Unapply Commit