Scalable Join Patterns

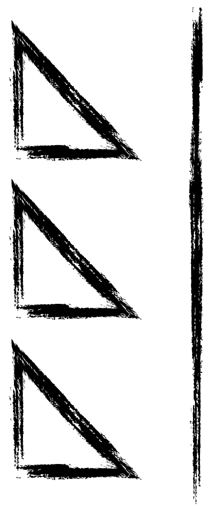
Aaron Turon
Northeastern University

Claudio Russo
Microsoft Research

Fine-grained parallelism requires communication

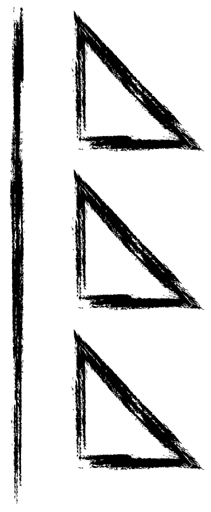
Fine-grained parallelism requires communication

Barrier



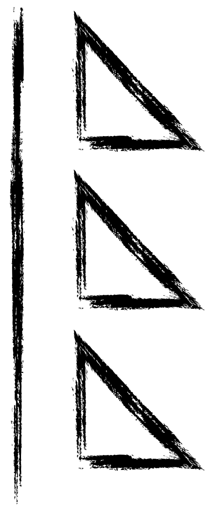
Fine-grained parallelism requires communication

Barrier

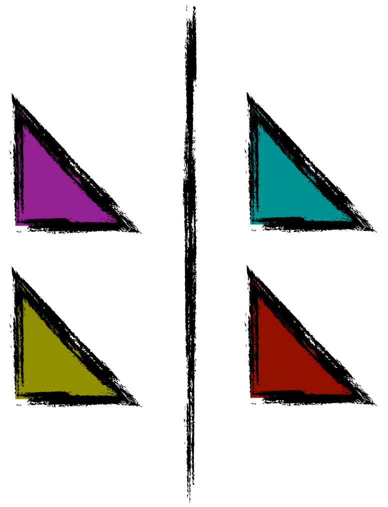


Fine-grained parallelism requires communication

Barrier

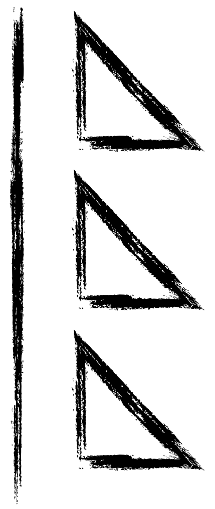


Exchanger

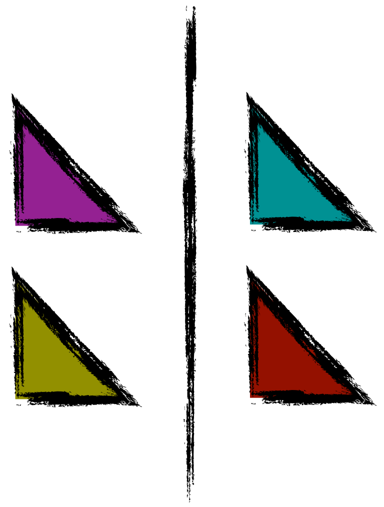


Fine-grained parallelism requires communication

Barrier



Exchanger



Lock

Fine-grained parallelism requires communication

Fine-grained parallelism requires communication

Synchronization protocols

Blocking & context switches

Cache coherence & memory bandwidth

State of the art?
Leave it to the experts:

$$\begin{tikzcd} \begin{array}{c}\text{Research}\\ \text{Literature}\end{array} \arrow[r, leftrightarrow] & \begin{array}{c}\text{Industrial-strength}\\ \text{Libraries}\end{array} \end{tikzcd}$$

$$\overbrace{\begin{array}{c}\text{java.util.concurrent}\\ \text{.NET 4.0}\\ \text{Intel TBB}\end{array}}$$

The problem

Libraries are an enormous undertaking, not extensible or customizable by users

This work

Use ***join patterns*** for synchronization: [Fournet & Gonthier]

Expressive

Write synchronization primitives declaratively and concisely

Scalable

Competitive with industrial libraries; can recover existing algorithms

Expressive

Joins: a crash course

```
class Lock {
  public Synchronous.Channel  Acquire = new ...
  public Asynchronous.Channel Release = new ...
  public Lock() {
    When(Acquire).And(Release).Do(() => {});

    // initially available
    Release();
  }
}
```

Joins: a crash course

```
class Lock {
  public Synchronous.Channel  Acquire = new ...
  public Asynchronous.Channel Release = new ...
  public Lock() {
    When(Acquire).And(Release).Do(() => {});

    // initially available
    Release();
  }
}
```

Join pattern

Joins: a crash course

```
class Lock {
  public Synchronous.Channel  Acquire = new ...
  public Asynchronous.Channel Release = new ...
  public Lock() {
    When(Acquire).And(Release).Do(() => {});

    // initially available
    Release();
  }
}
```

$\uparrow$

Join body

Joins: a crash course

```
class Lock {
  public Synchronous.Channel  Acquire = new ...
  public Asynchronous.Channel Release = new ...
  public Lock() {
    When(Acquire).And(Release).Do(() => {});

    // initially available
    Release();
  }
}
```

Message send

Joins: a crash course

```
class Semaphore {
  public Synchronous.Channel  Acquire = new ...
  public Asynchronous.Channel Release = new ...
  public Semaphore(int n) {
    When(Acquire).And(Release).Do(() => {});

    // initially n available
    for (; n > 0; n--) Release();
  }
}
```

Joins: a crash course

```
class ProducerConsumer<T> {
  public Synchronous<T>.Channel  Get = new ...
  public Asynchronous.Channel<T> Put = new ...
  public ProducerConsumer() {
    When(Get).And(Put).Do(t => t);
  }
}
```

Joins: a crash course

```
class ProducerConsumer<T> {
  public Synchronous<T>.Channel  Get = new ...
  public Asynchronous.Channel<T> Put = new ...
  public ProducerConsumer() {
    When(Get).And(Put).Do(t => t);
  }
}
```

Joins: a crash course

```
class ProducerConsumer<T> {
  public Synchronous<T>.Channel  Get = new ...
  public Asynchronous.Channel<T> Put = new ...
  public ProducerConsumer() {
    When(Get).And(Put).Do(t => t);
  }
}
```

Joins: a crash course

```
Synchronous.Channel[] hungry = new ...
Asynchronous.Channel[] chopstick = new ...

for (int i = 0; i < n; i++) {
  var left  = chopstick[i];
  var right = chopstick[(i+1) % n];
  When(hungry[i]).And(left).And(right).Do(() => {
    eat(); left(); right();
  });
}
```

Scalable

Existing joins implementations

Coarse-grained locks:

- Serialized matching
- Extra memory bus traffic

Our implementation (in C#)

Key idea:

Messages are resources

Our implementation (in C#)

Key idea:

Messages are resources

Store in lock-free bags

➡ parallelized matching

➡ decreased communication

Our implementation (in C#)

Key idea:

Messages are resources

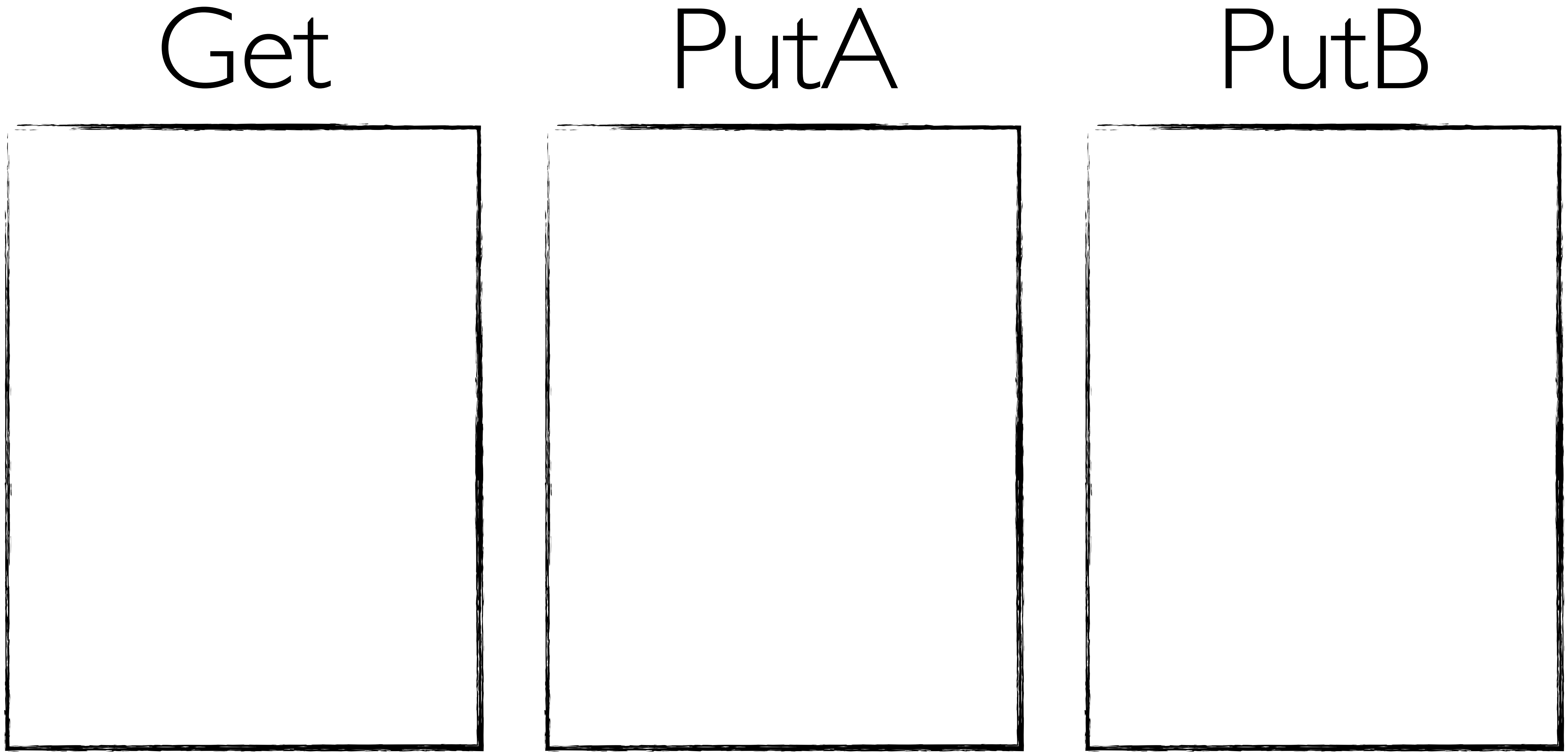
Store in lock-free bags

➡ parallelized matching

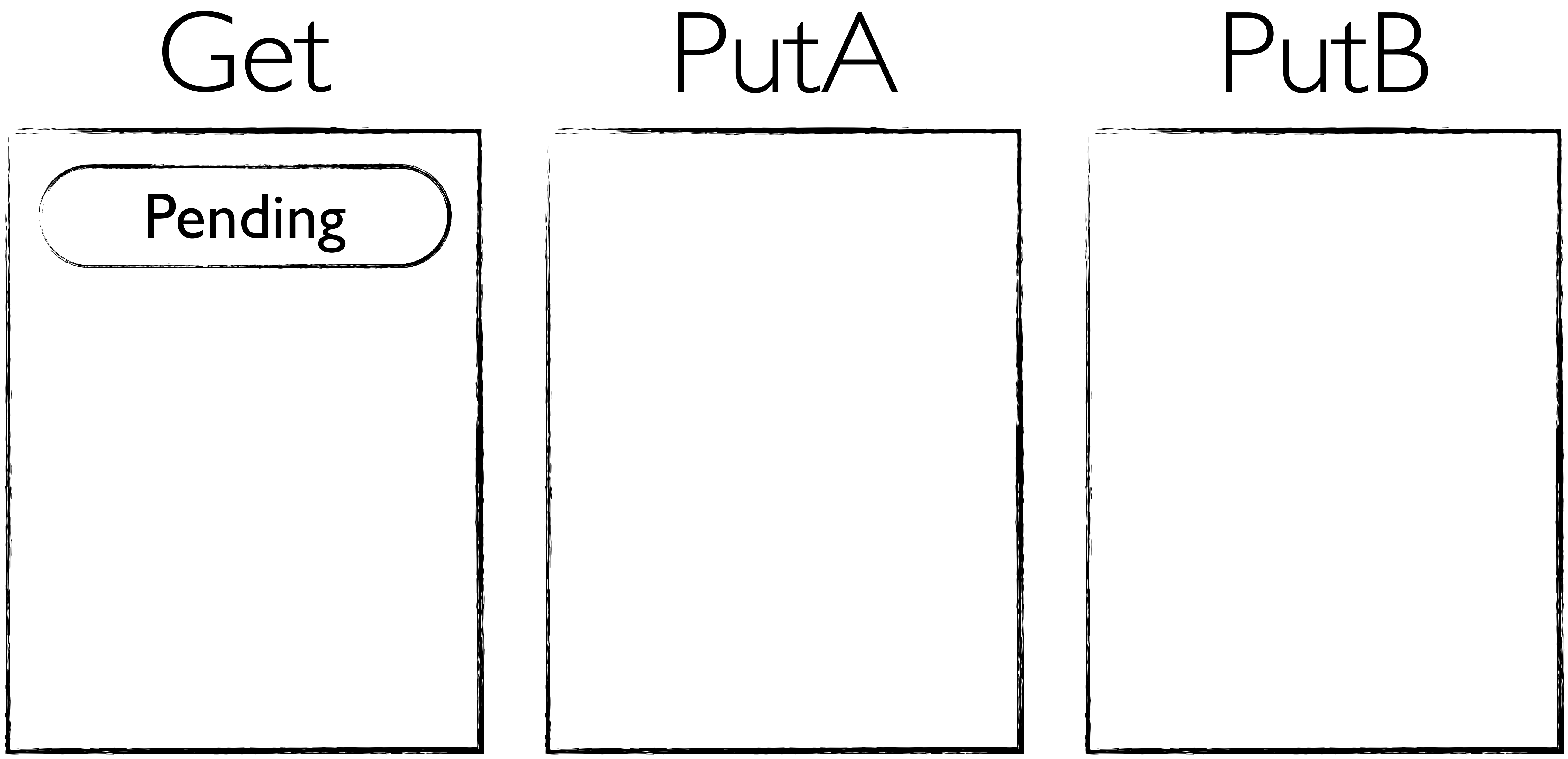
➡ decreased communication

Acquire in global order

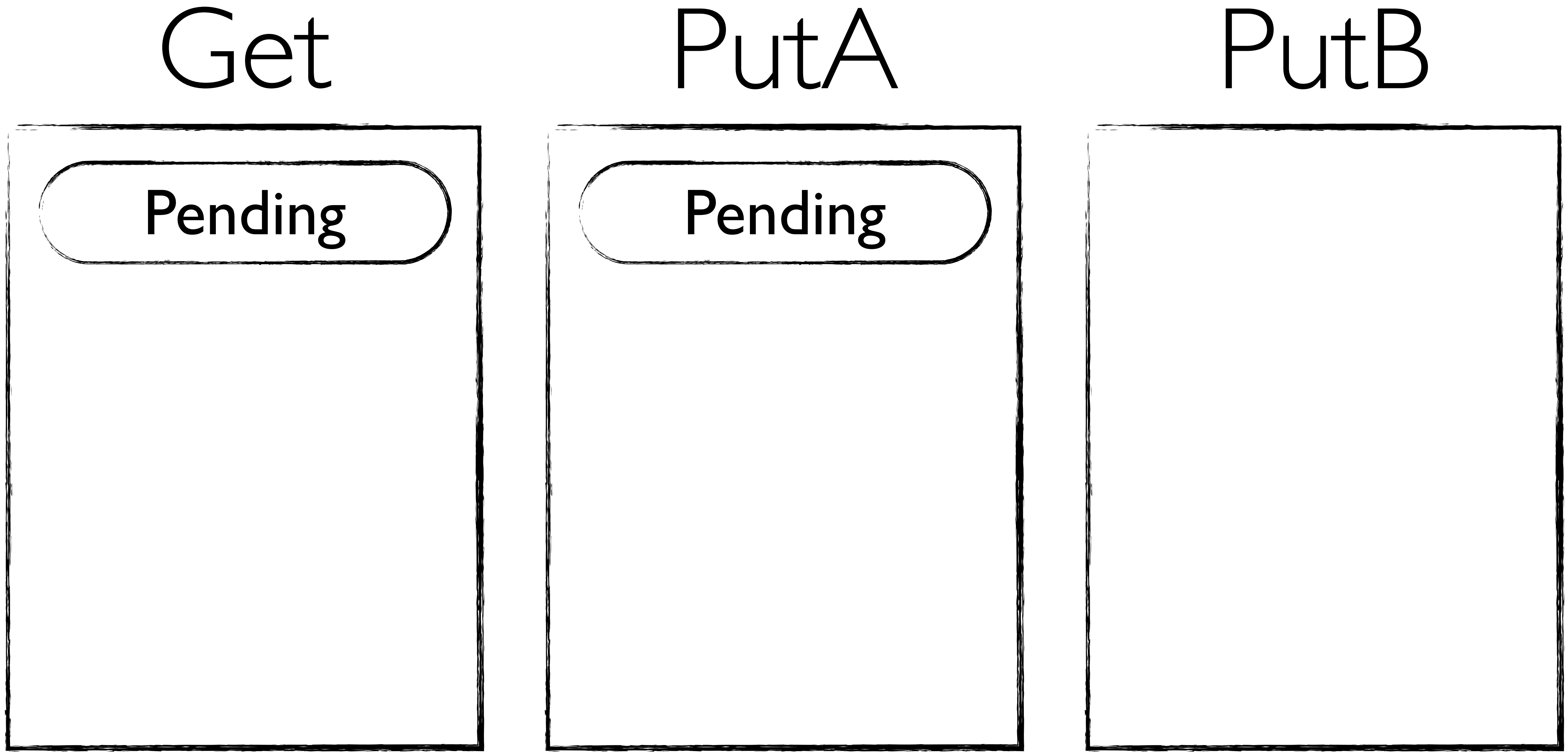
```
When(Get).And(PutA).And(PutB).Do(...)
When(Get).And(PutA).And(PutB).Do(...)
```



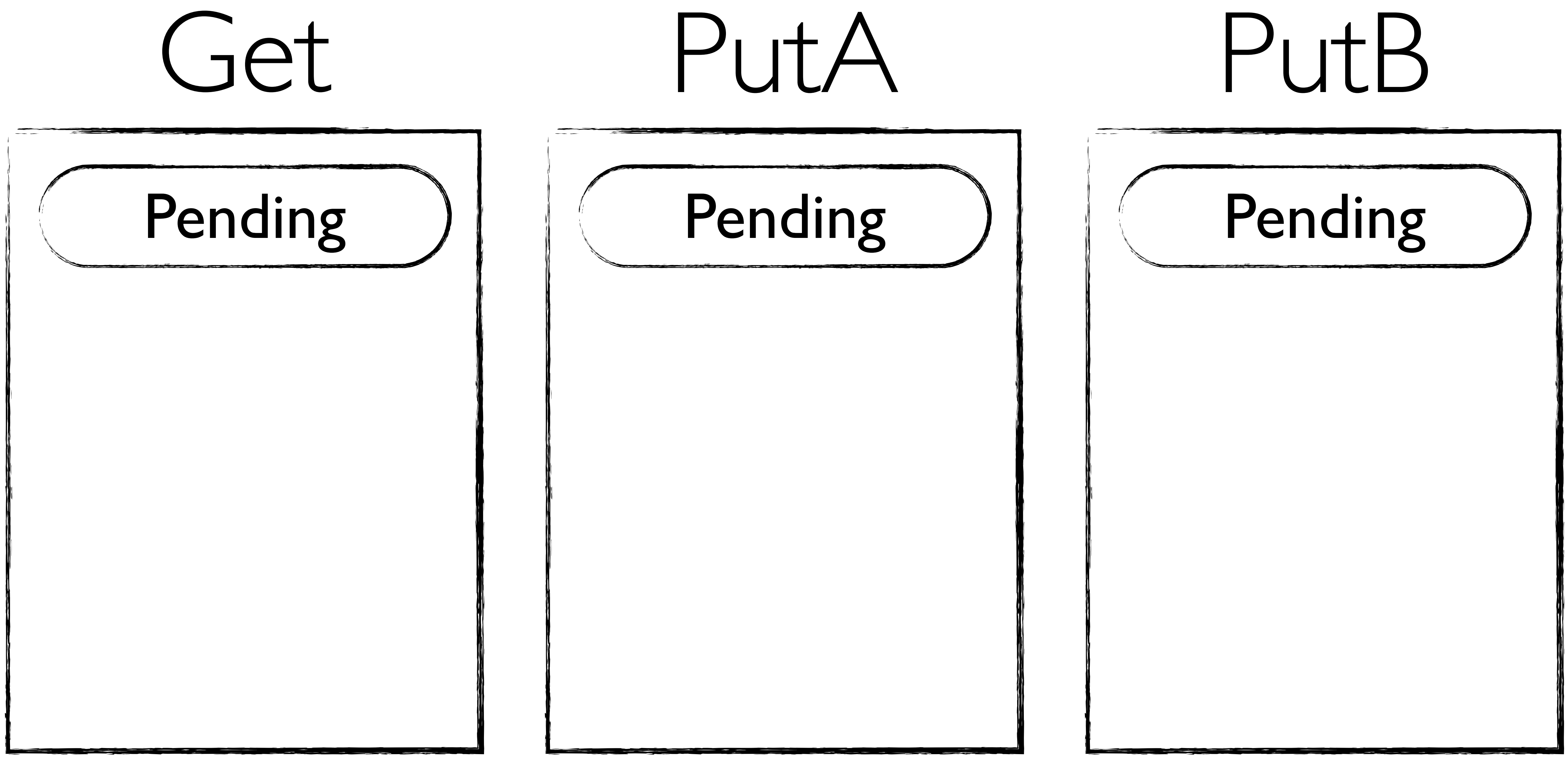
```
When(Get).And(PutA).And(PutB).Do(...)
```



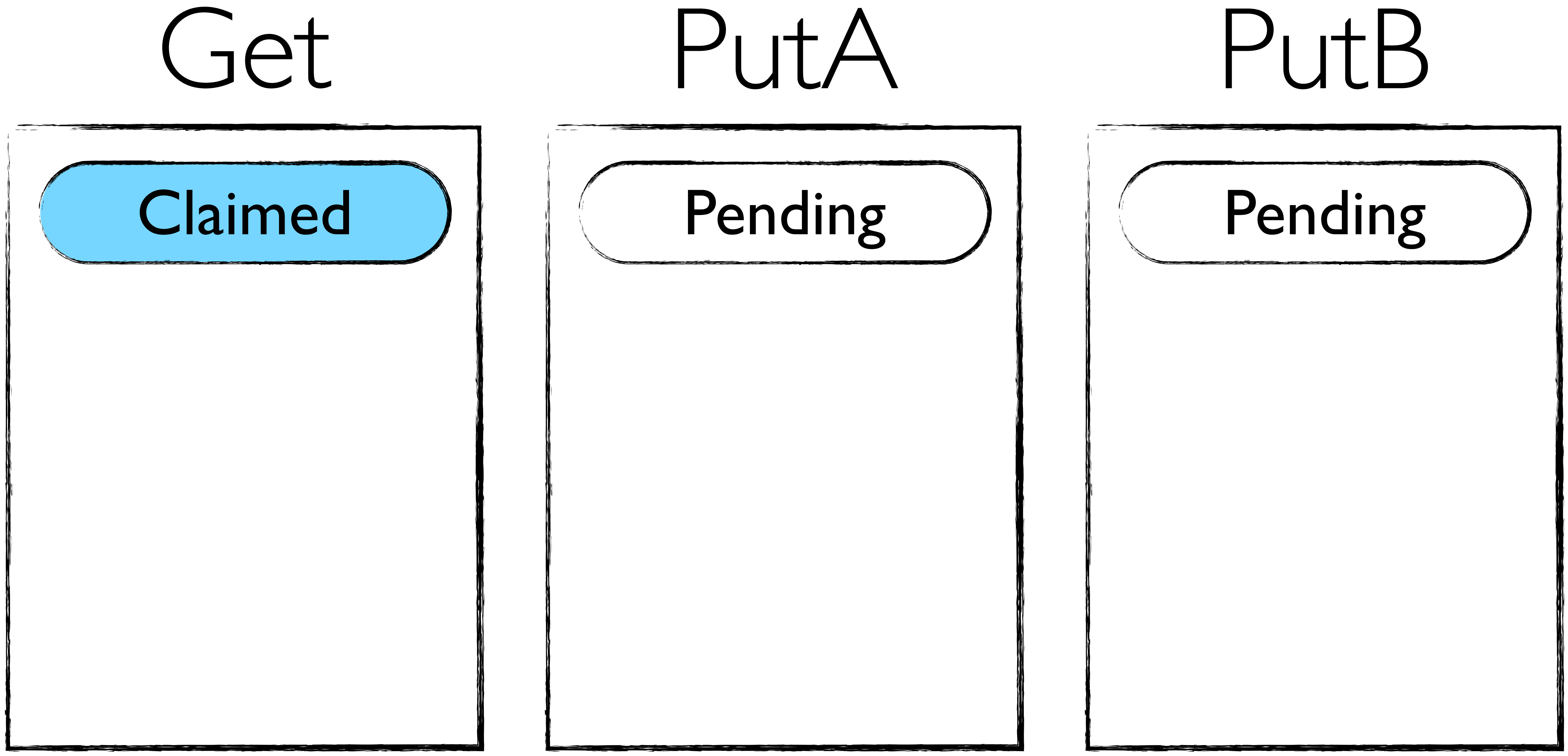
```
When(Get).And(PutA).And(PutB).Do(...)
```



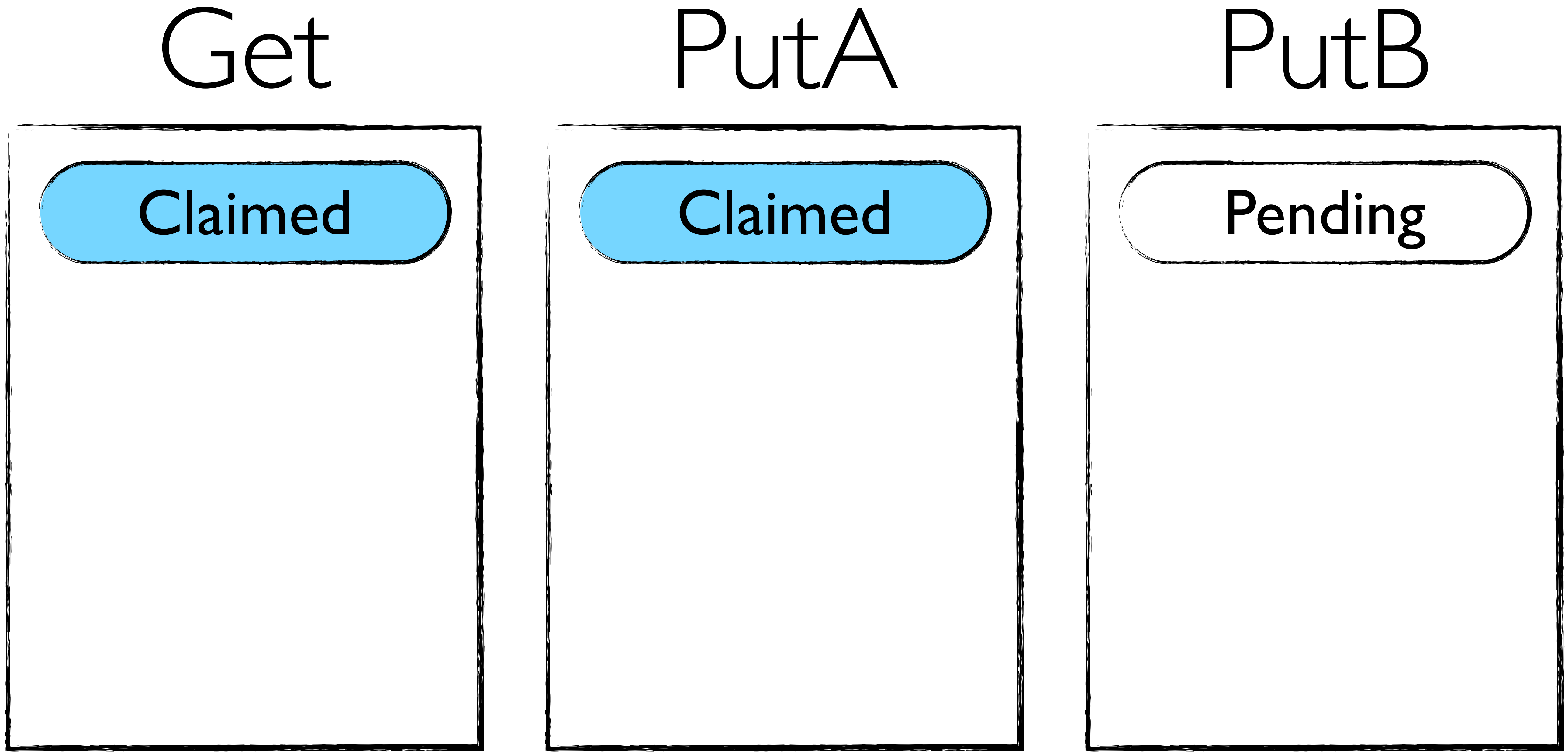
```
When(Get).And(PutA).And(PutB).Do(...)
```



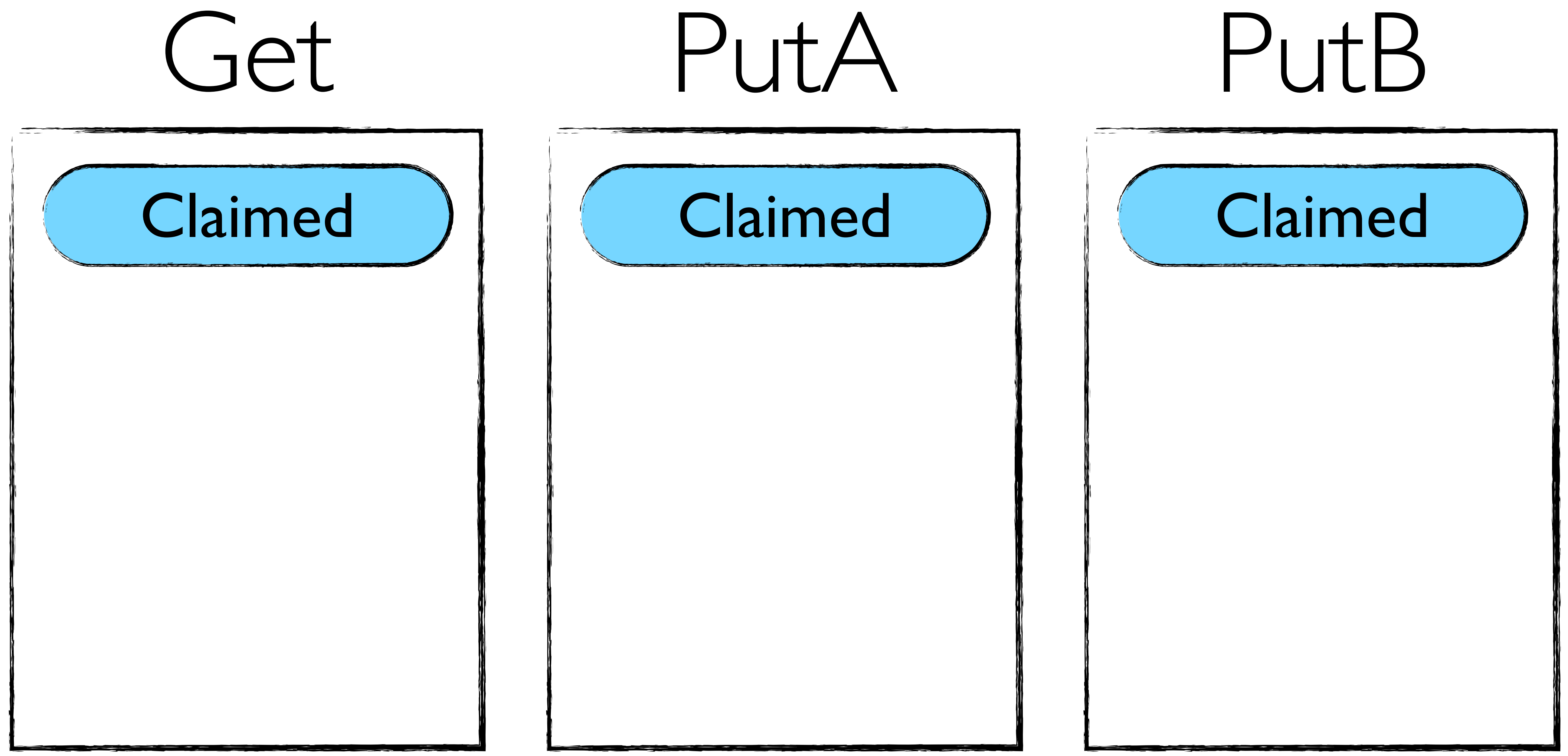
`When(Get).And(PutA).And(PutB).Do(...)`



```
When(Get).And(PutA).And(PutB).Do(...)
```



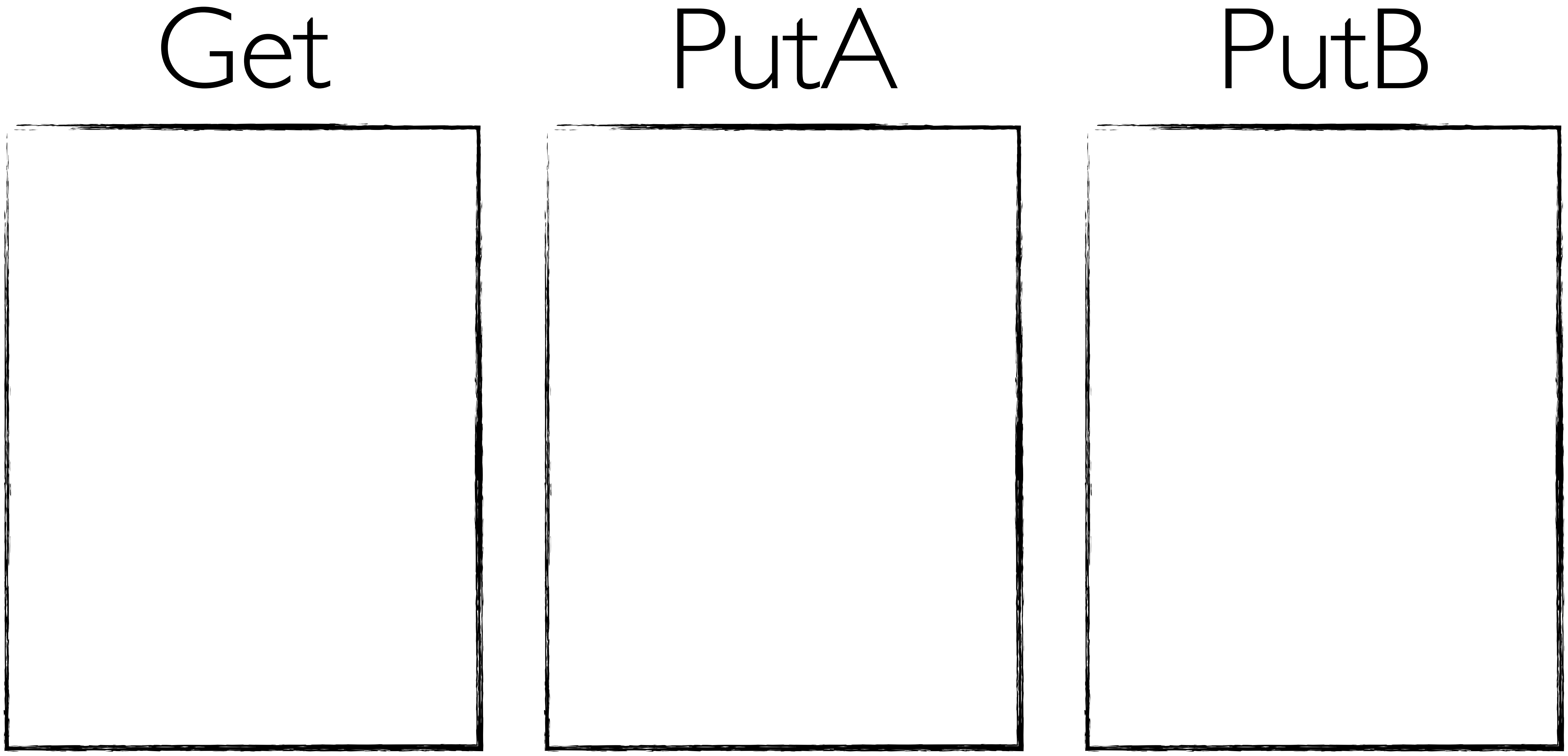
```
When(Get).And(PutA).And(PutB).Do(...)
```



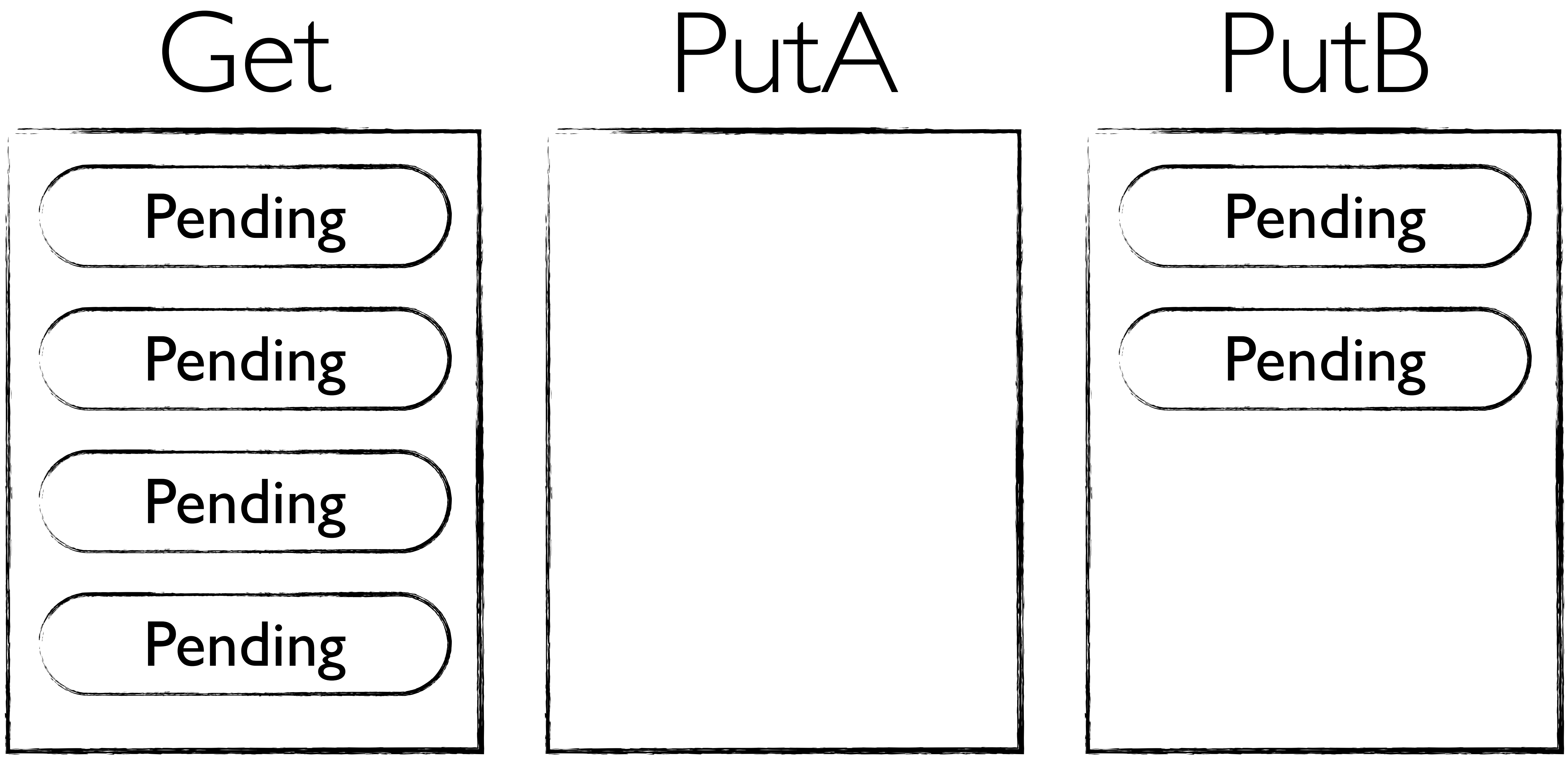
```
When(Get).And(PutA).And(PutB).Do(...)
```

Get	PutA	PutB
Consumed	Consumed	Consumed

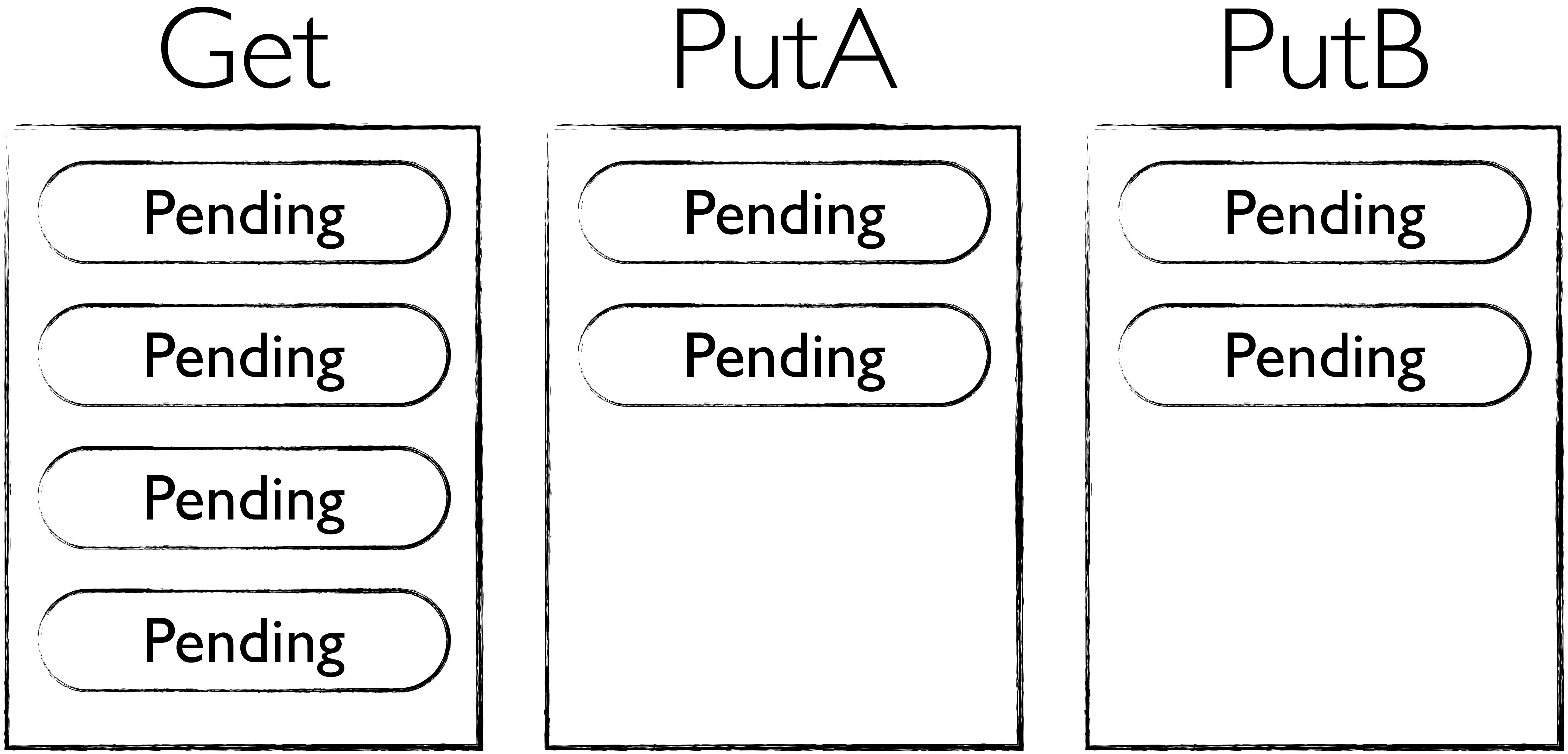
```
When(Get).And(PutA).And(PutB).Do(...)
```



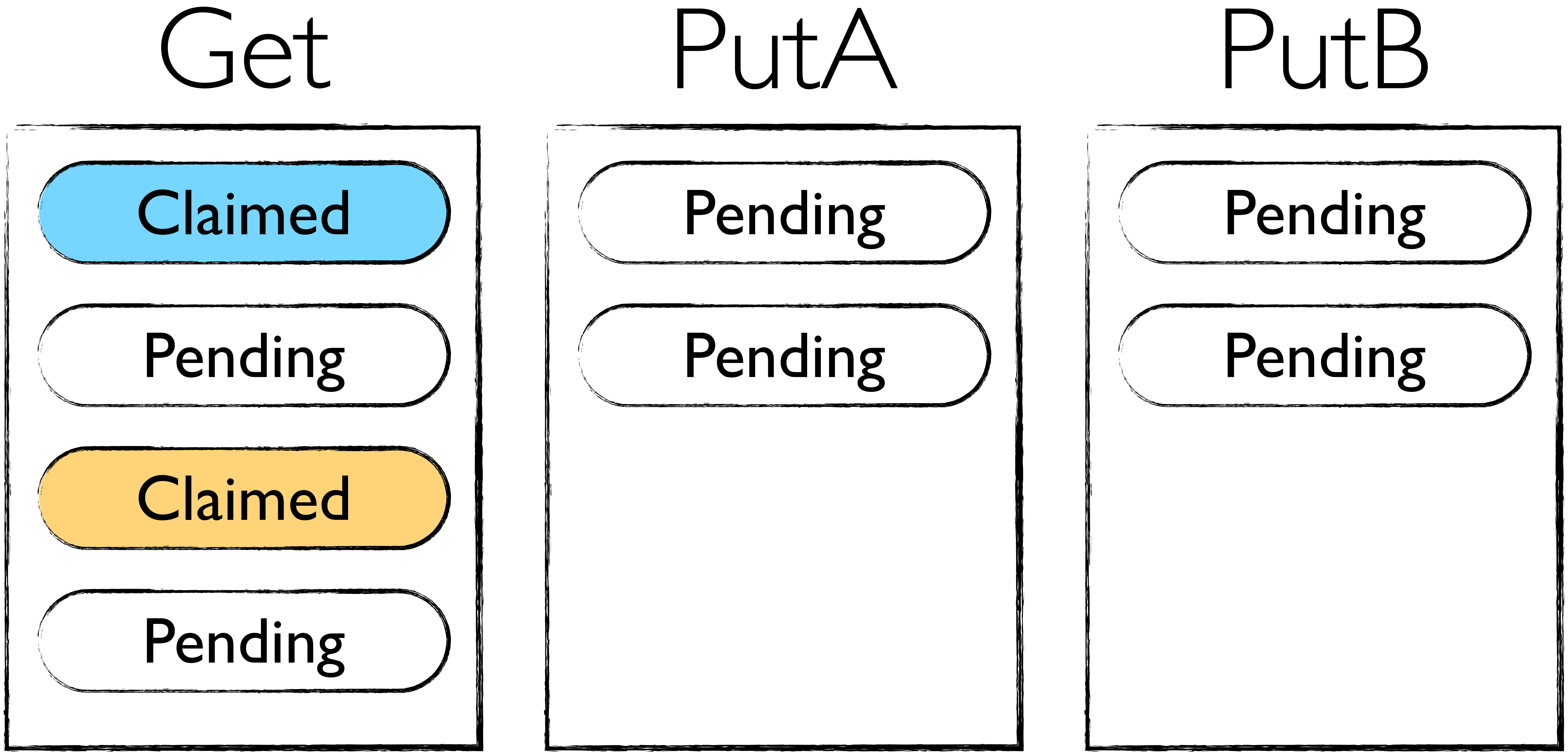
```
When(Get).And(PutA).And(PutB).Do(...)
```



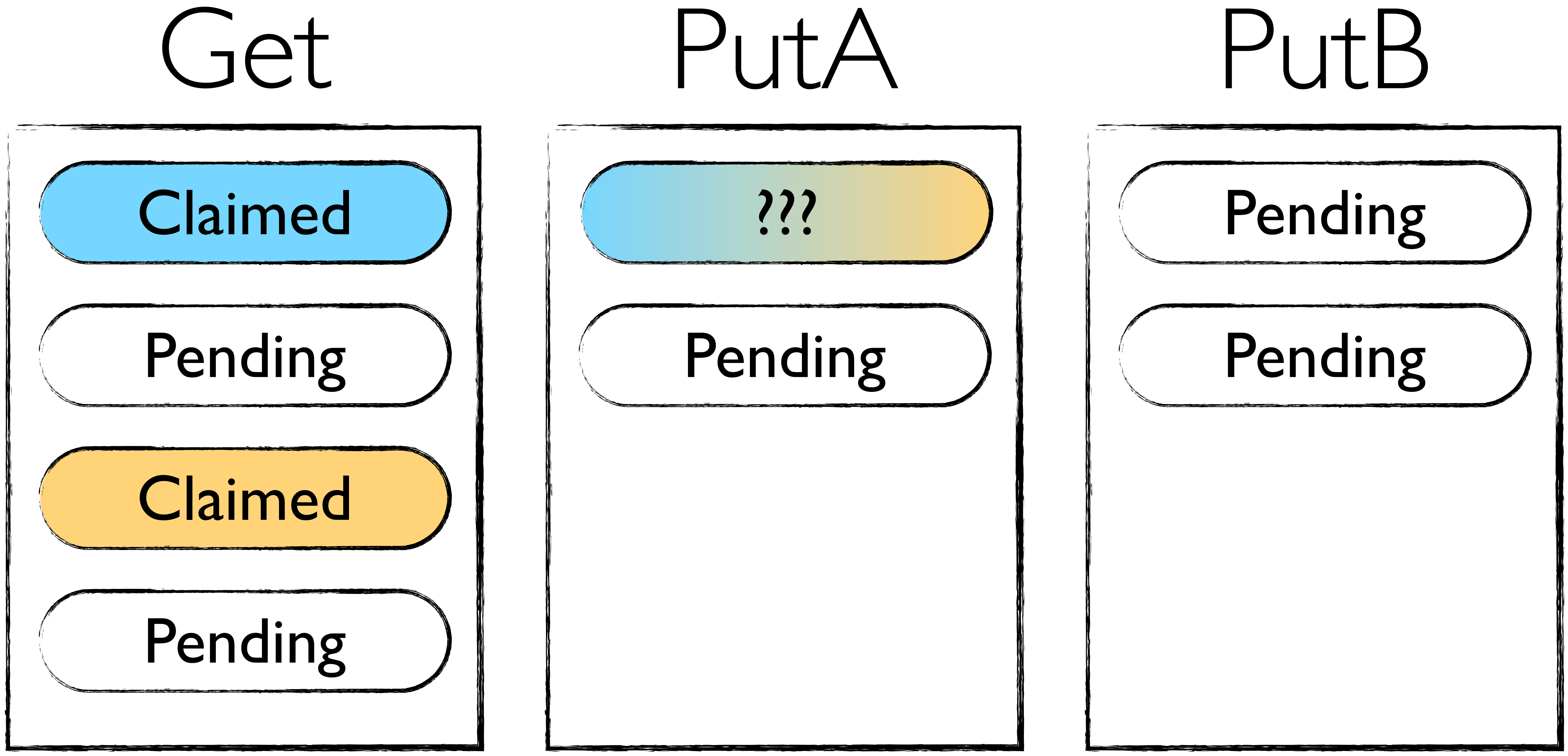
```
When(Get).And(PutA).And(PutB).Do(...)
```



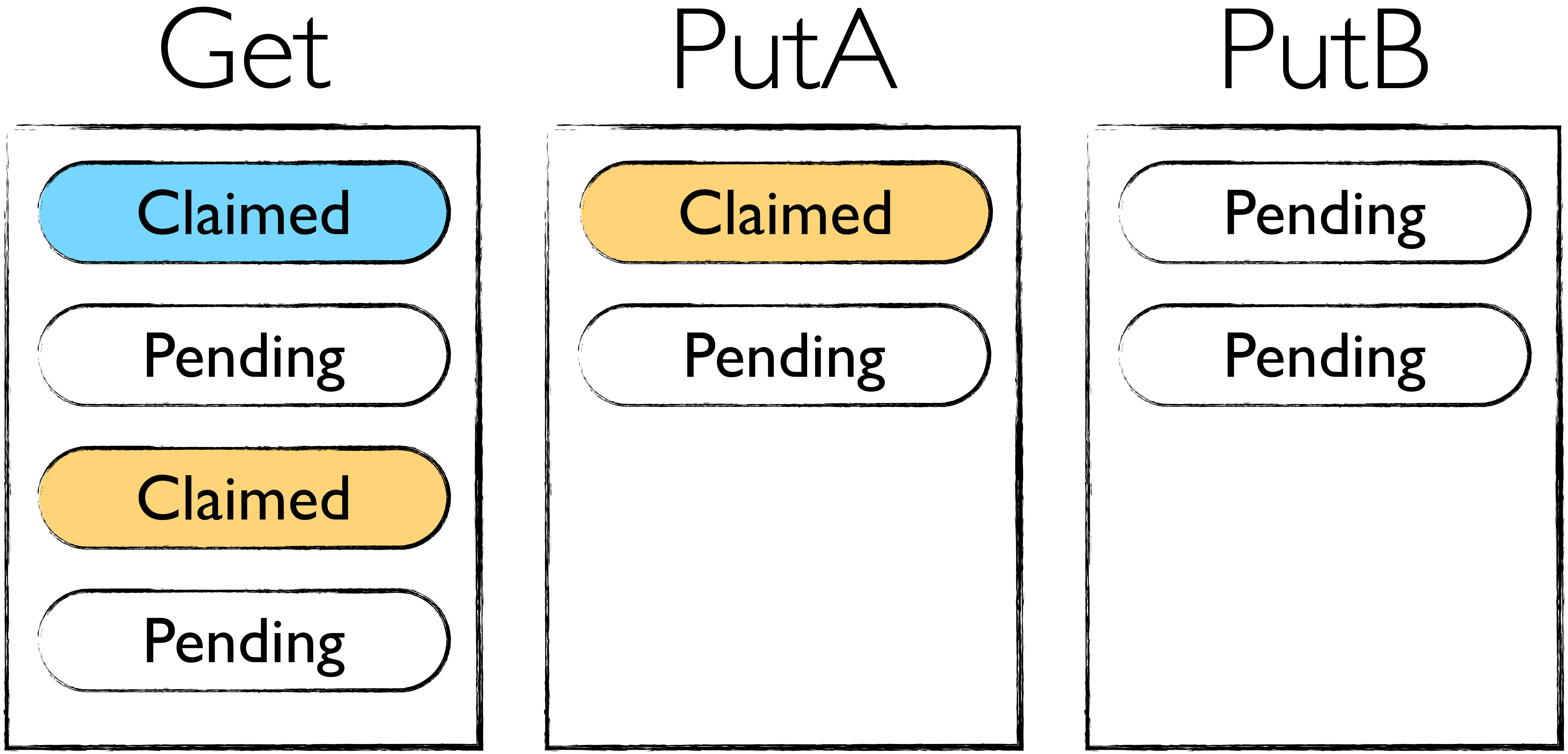
`When(Get).And(PutA).And(PutB).Do(...)`



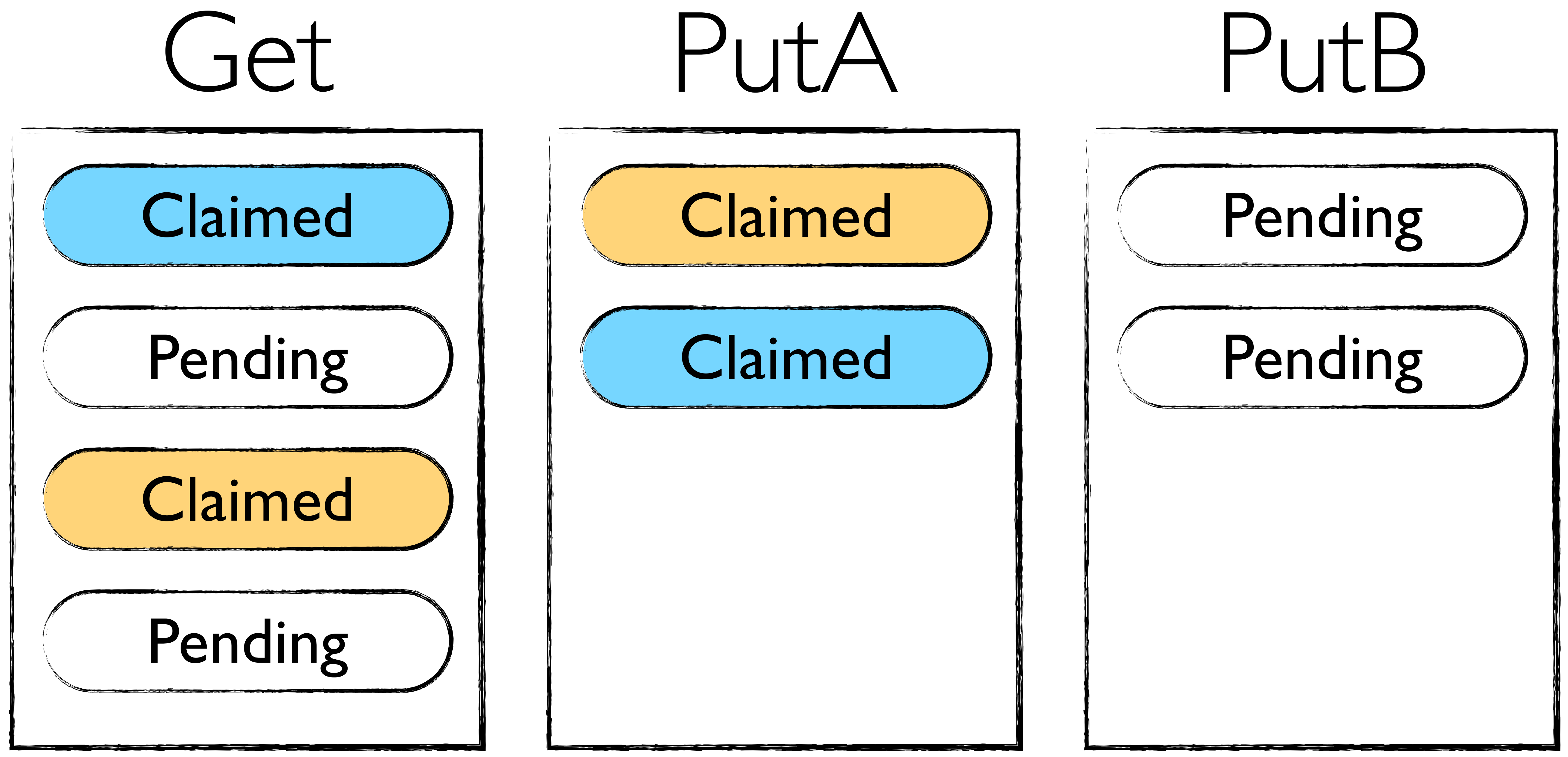
`When(Get).And(PutA).And(PutB).Do(...)`



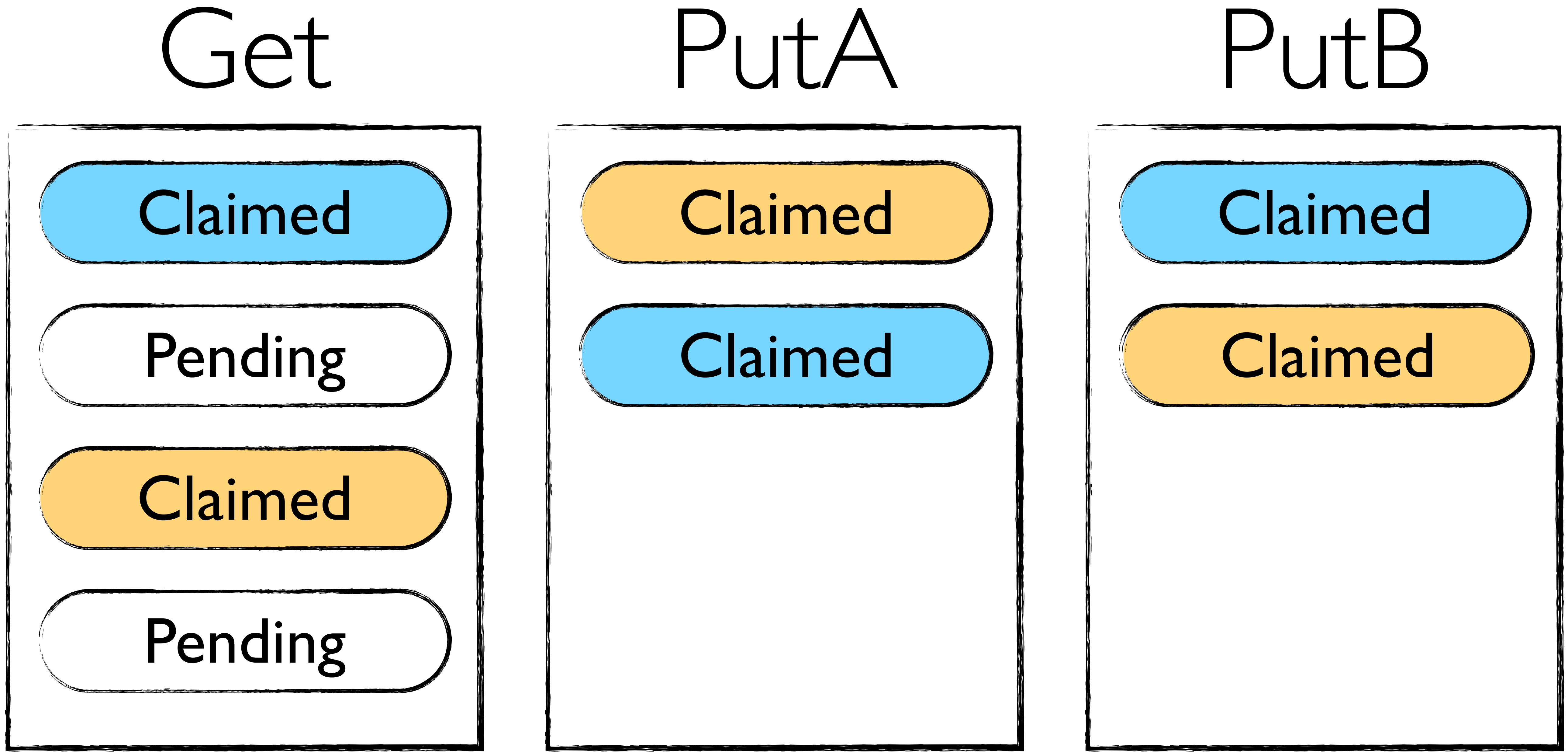
```
When(Get).And(PutA).And(PutB).Do(...)
```



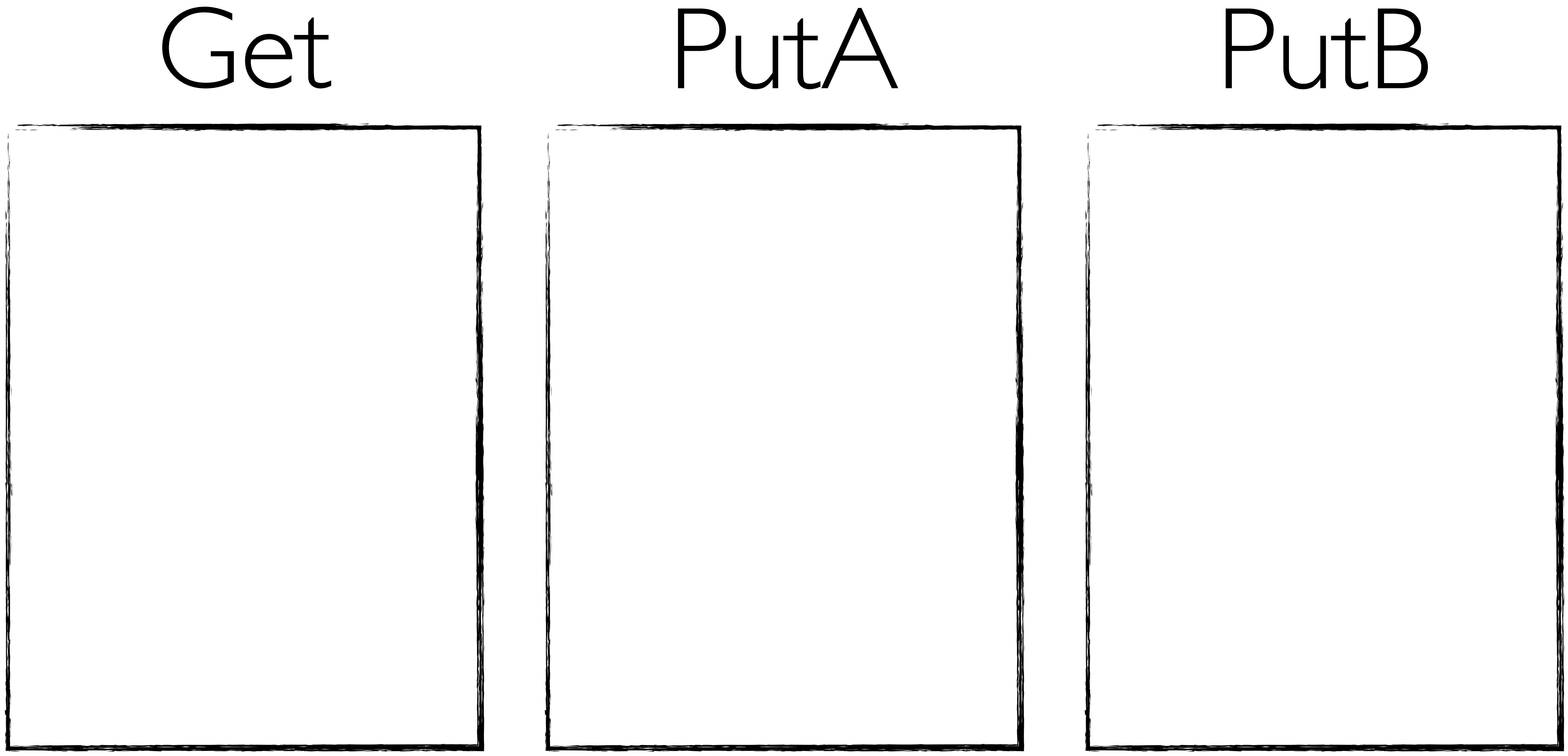
```
When(Get).And(PutA).And(PutB).Do(...)
```



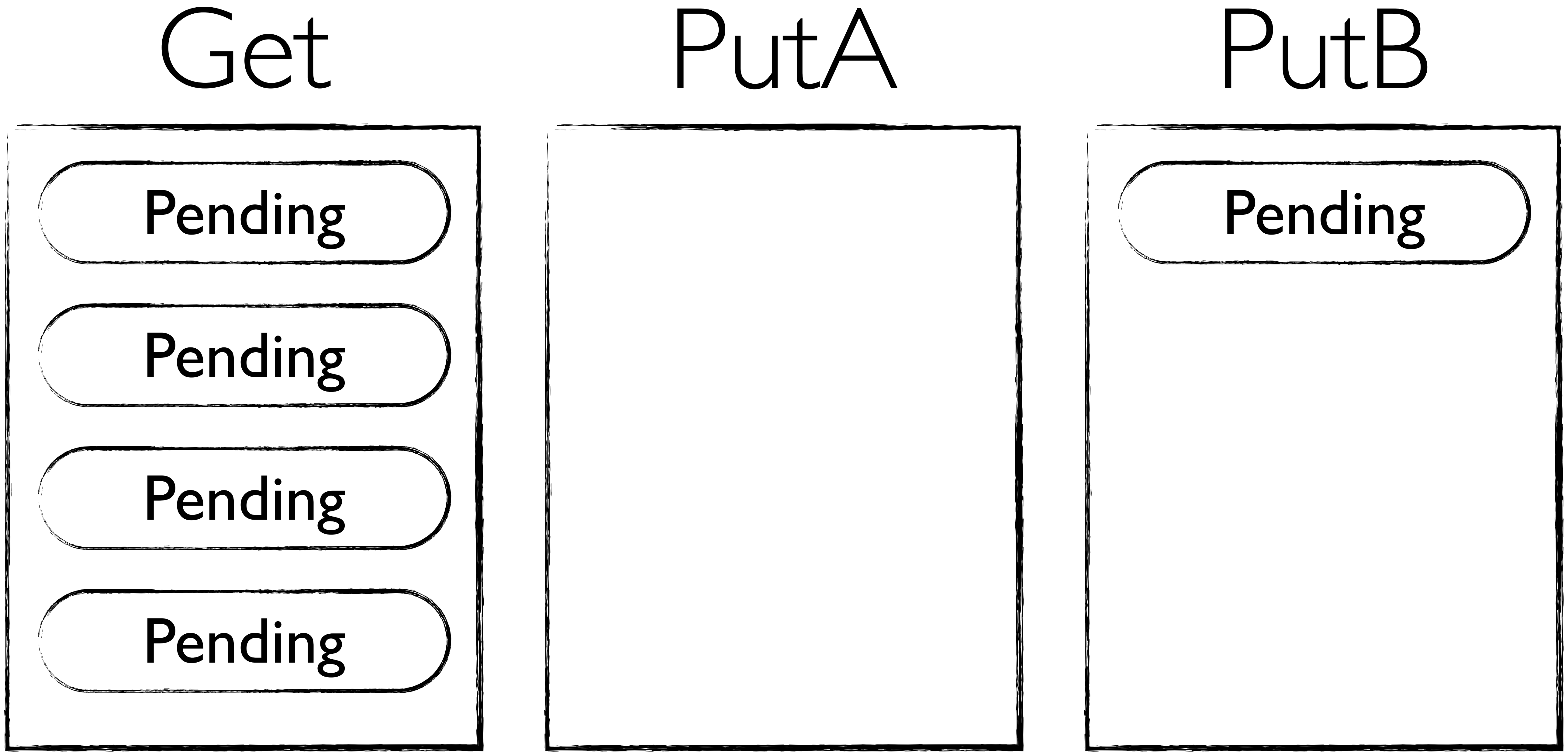
```
When(Get).And(PutA).And(PutB).Do(...)
```



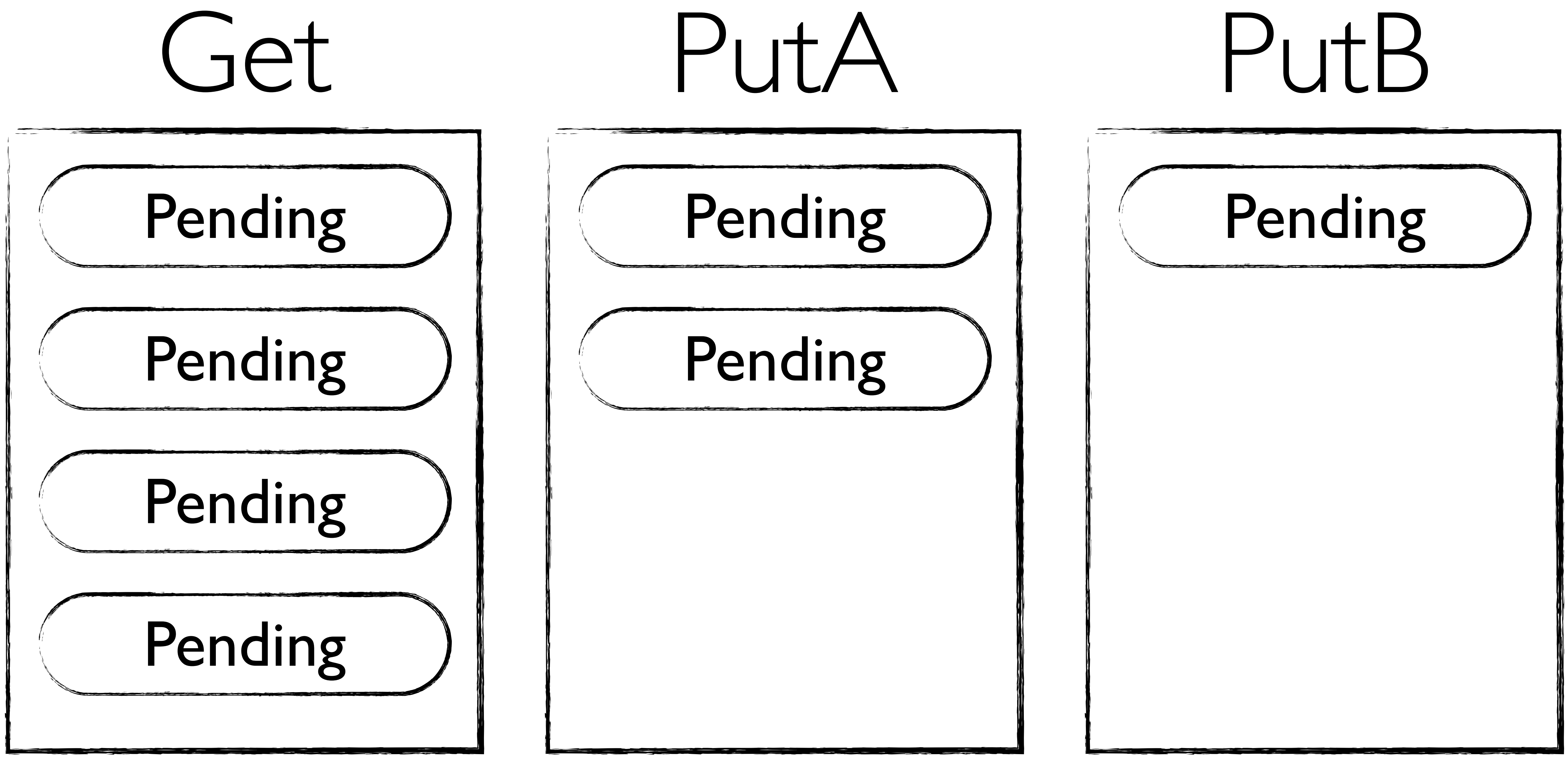
```
When(Get).And(PutA).And(PutB).Do(...)
```



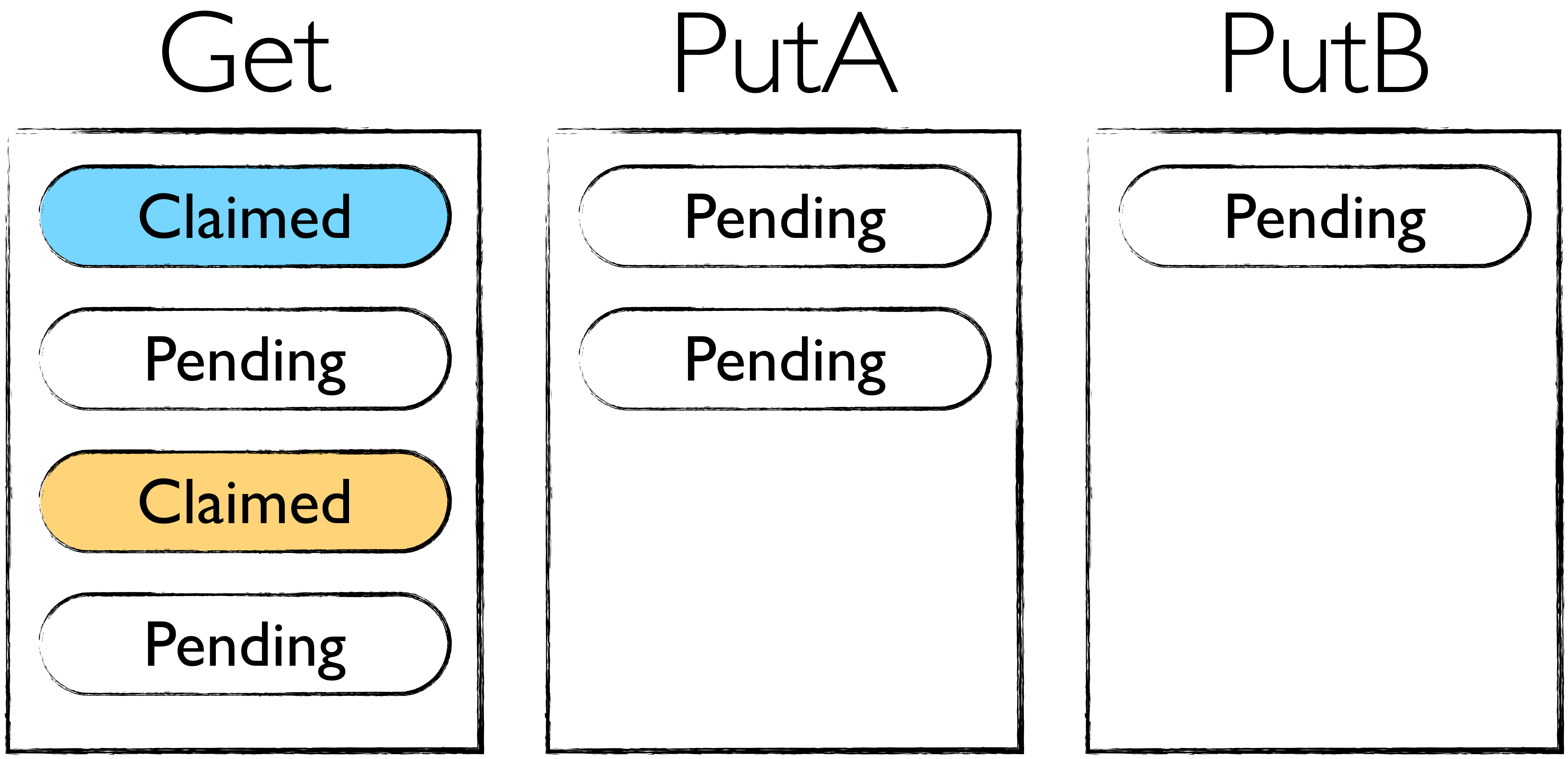
```
When(Get).And(PutA).And(PutB).Do(...)
```



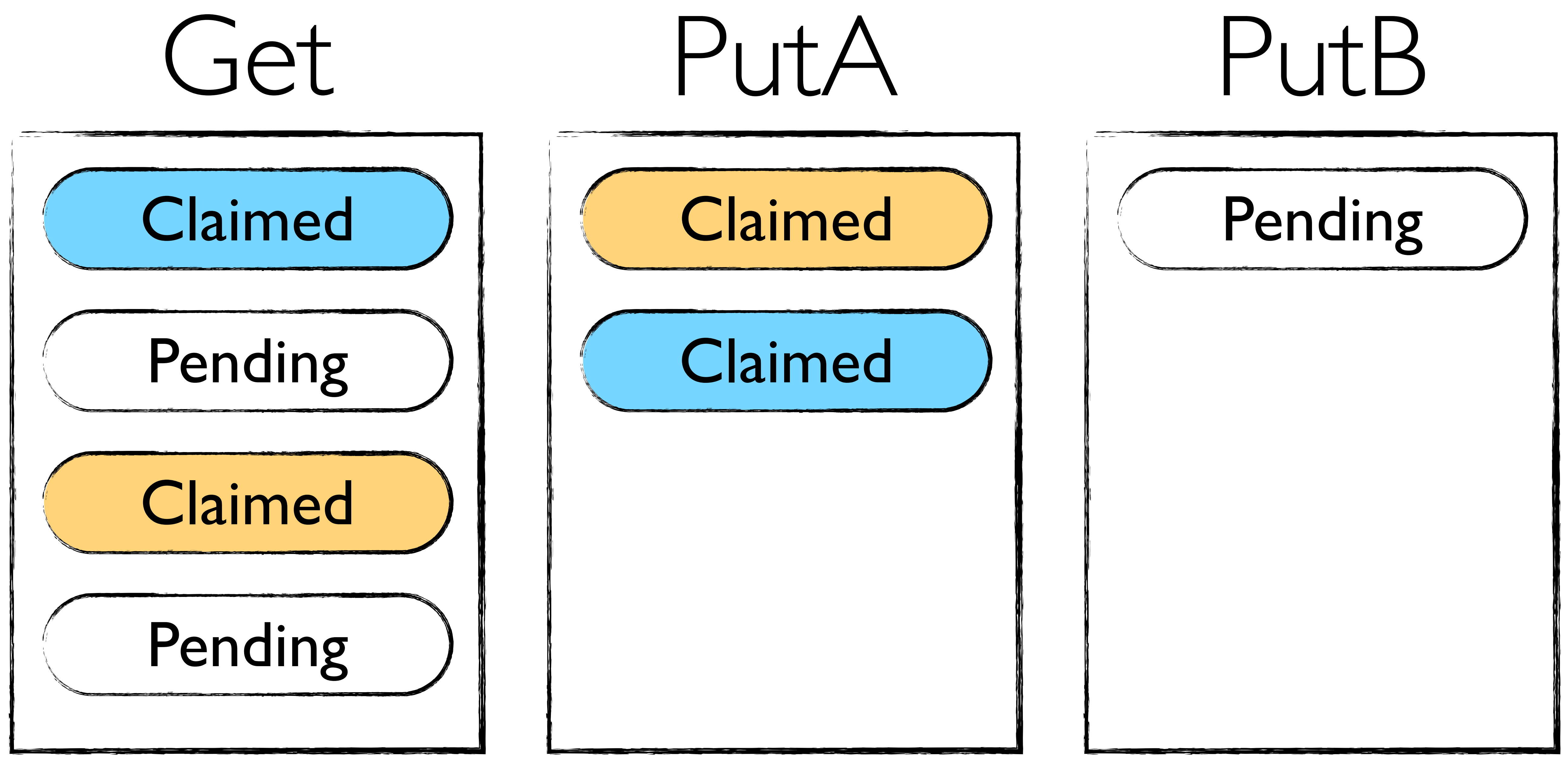
`When(Get).And(PutA).And(PutB).Do(...)`



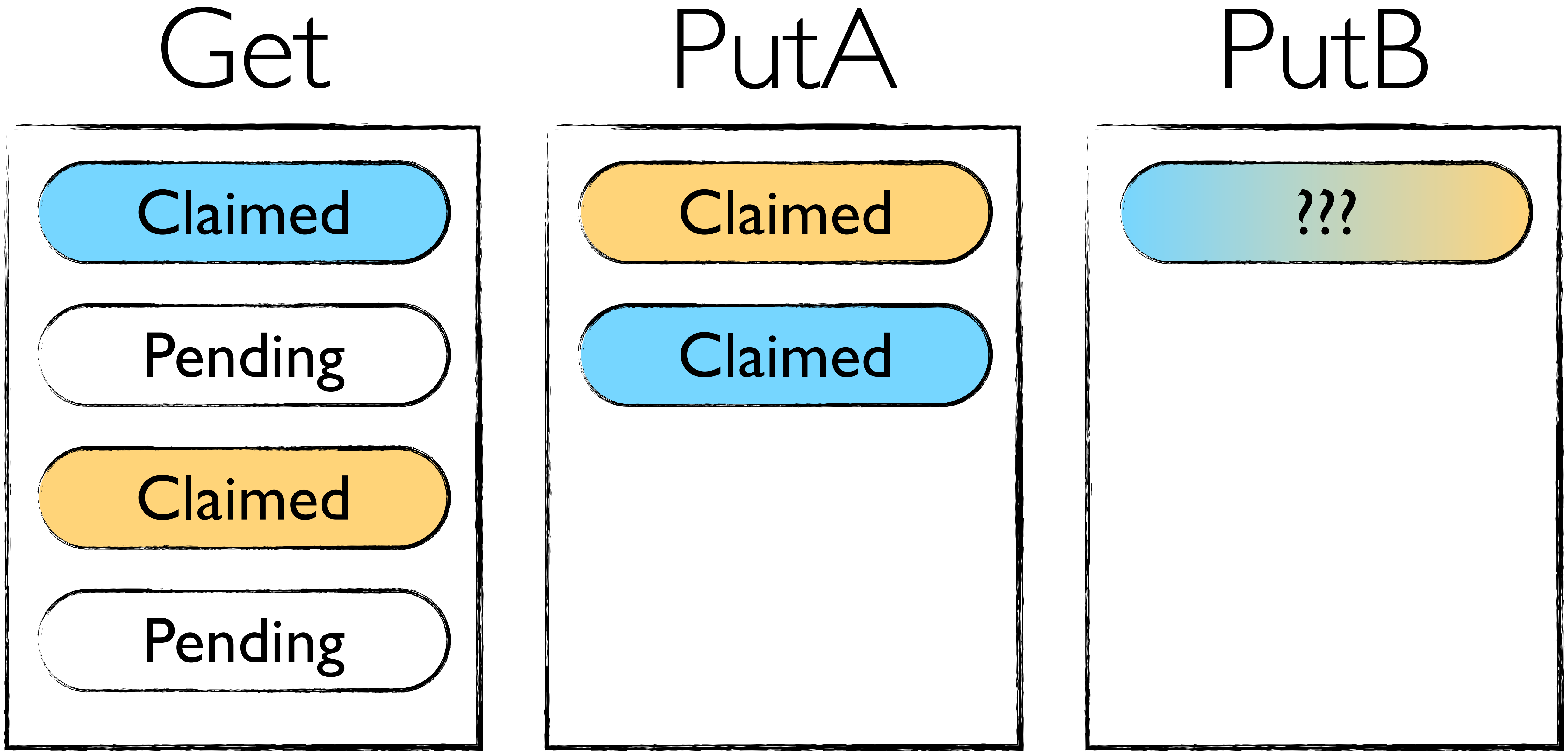
```
When(Get).And(PutA).And(PutB).Do(...)
```



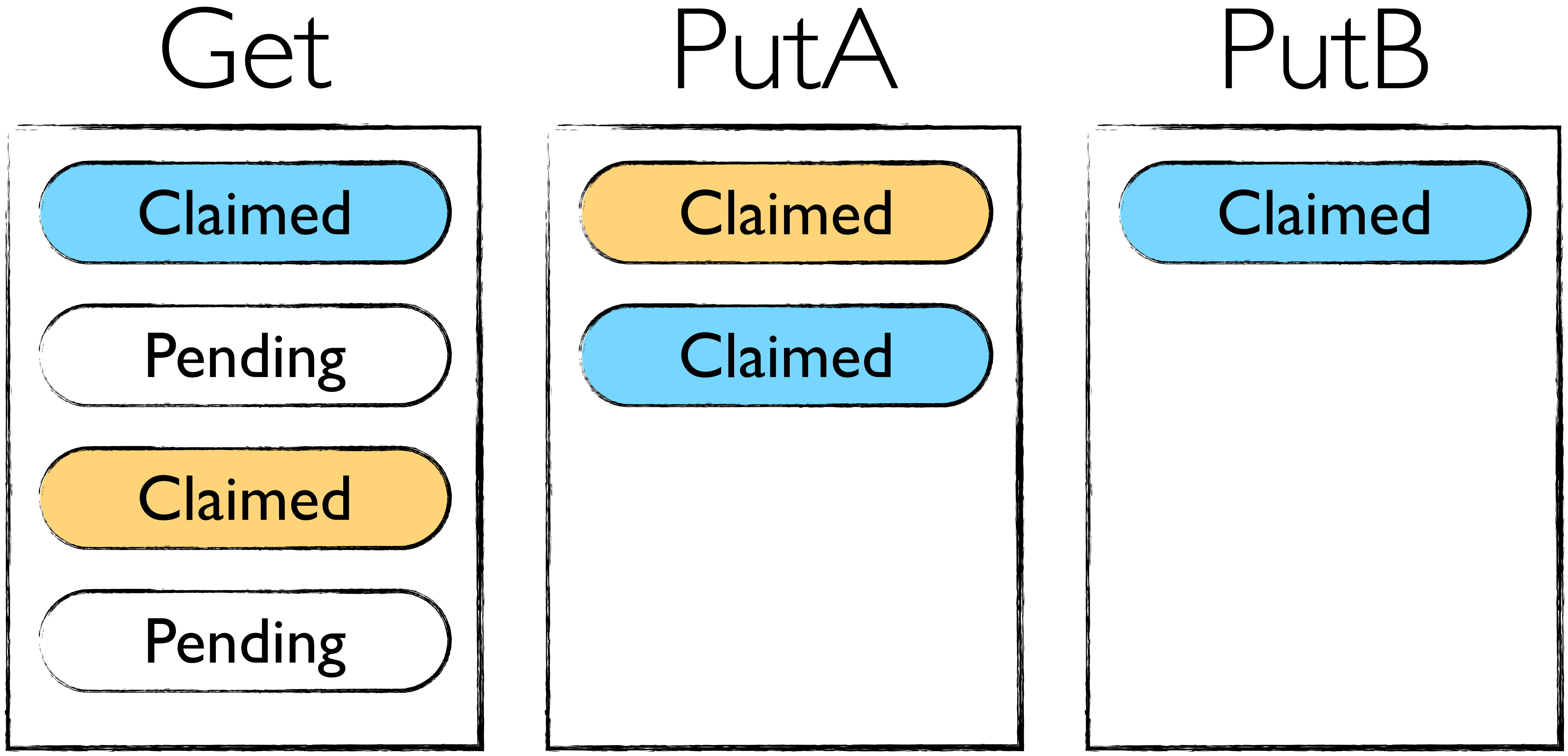
```
When(Get).And(PutA).And(PutB).Do(...)
```



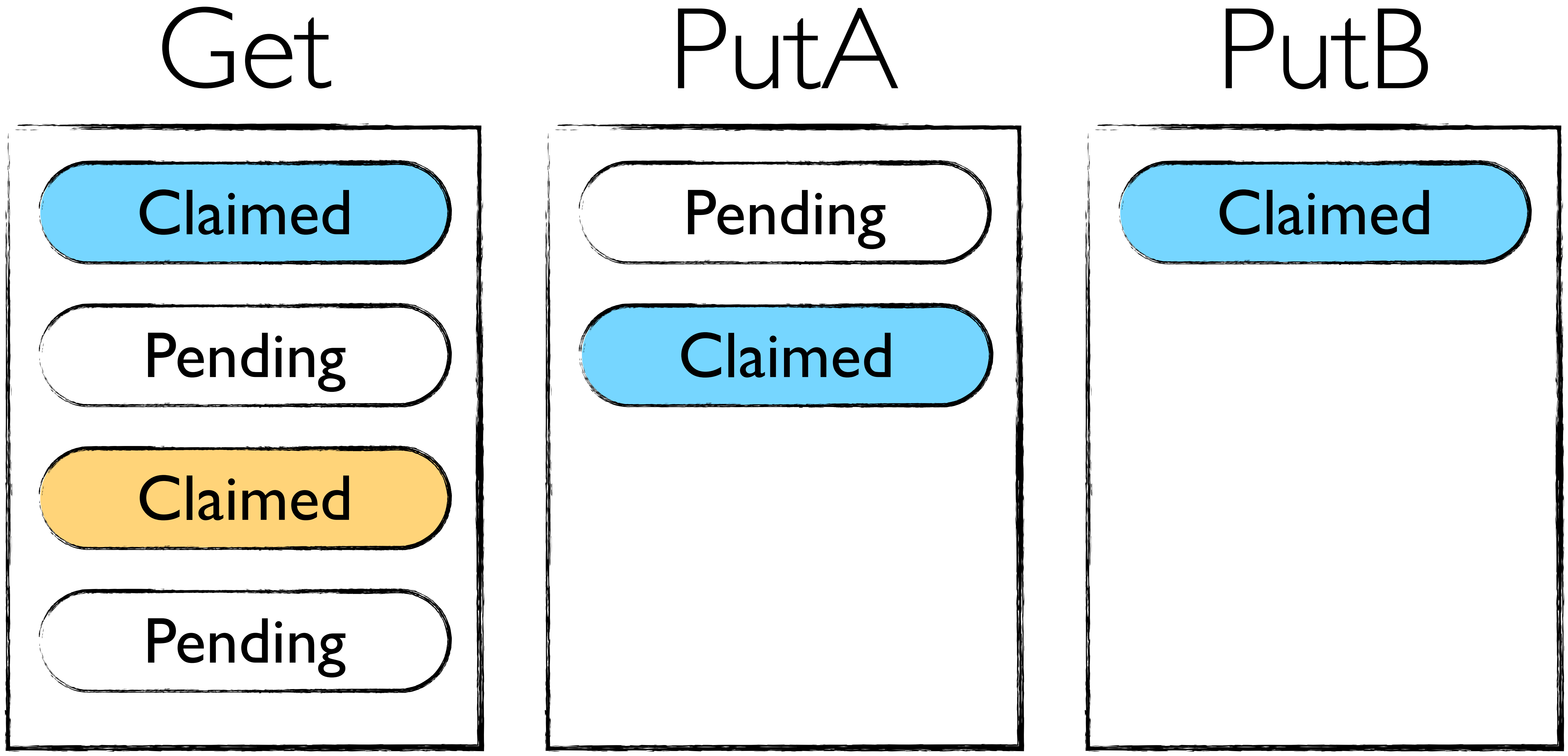
```
When(Get).And(PutA).And(PutB).Do(...)
```



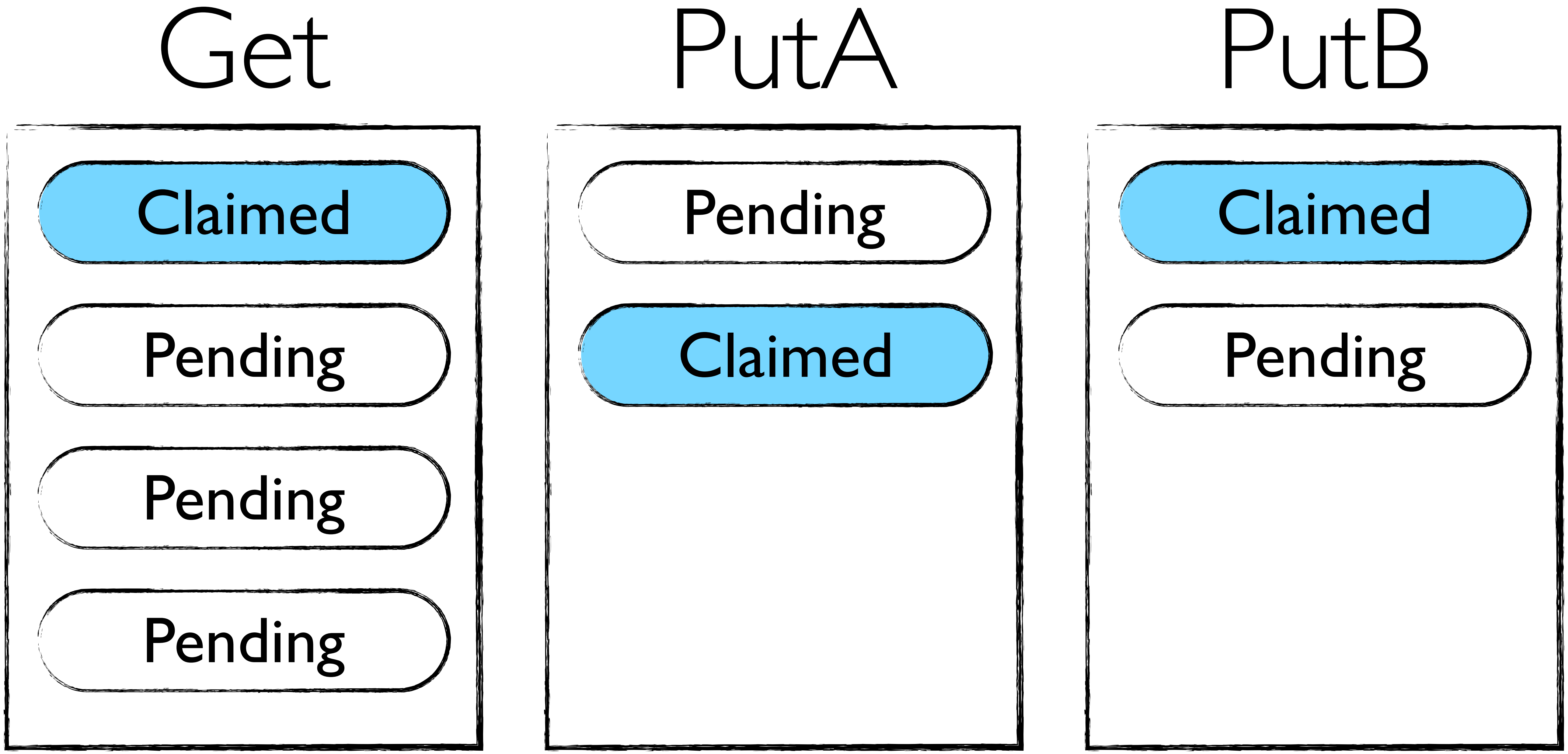
```
When(Get).And(PutA).And(PutB).Do(...)
```



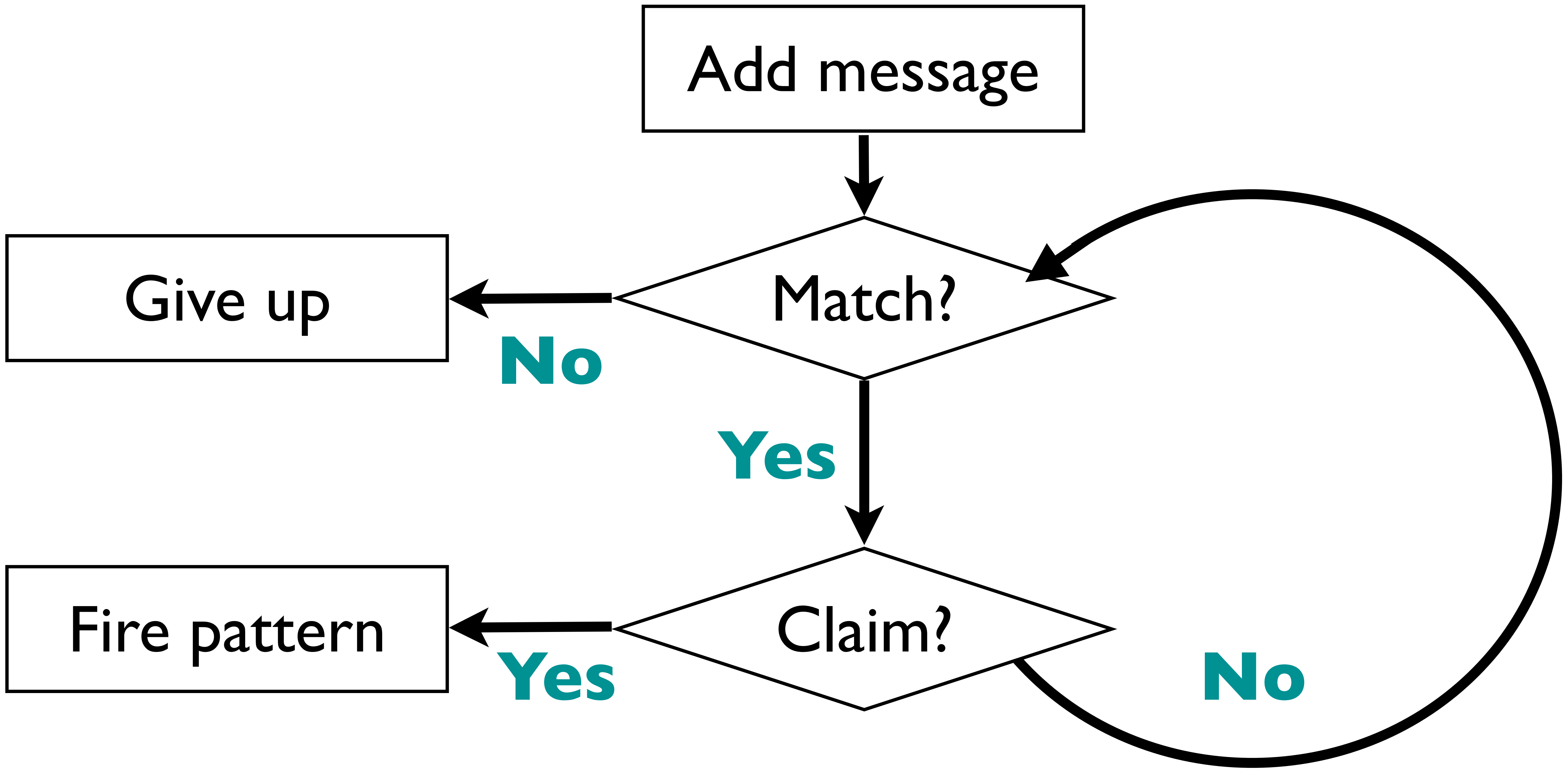
`When(Get).And(PutA).And(PutB).Do(...)`



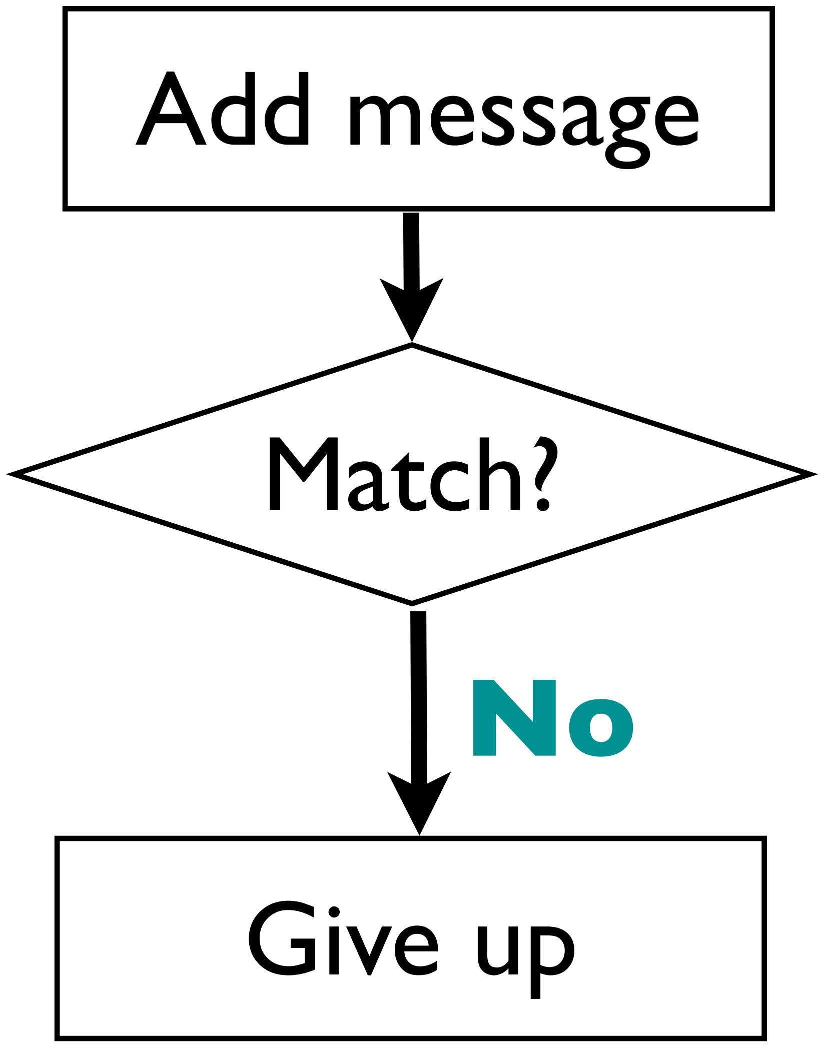
```
When(Get).And(PutA).And(PutB).Do(...)
```



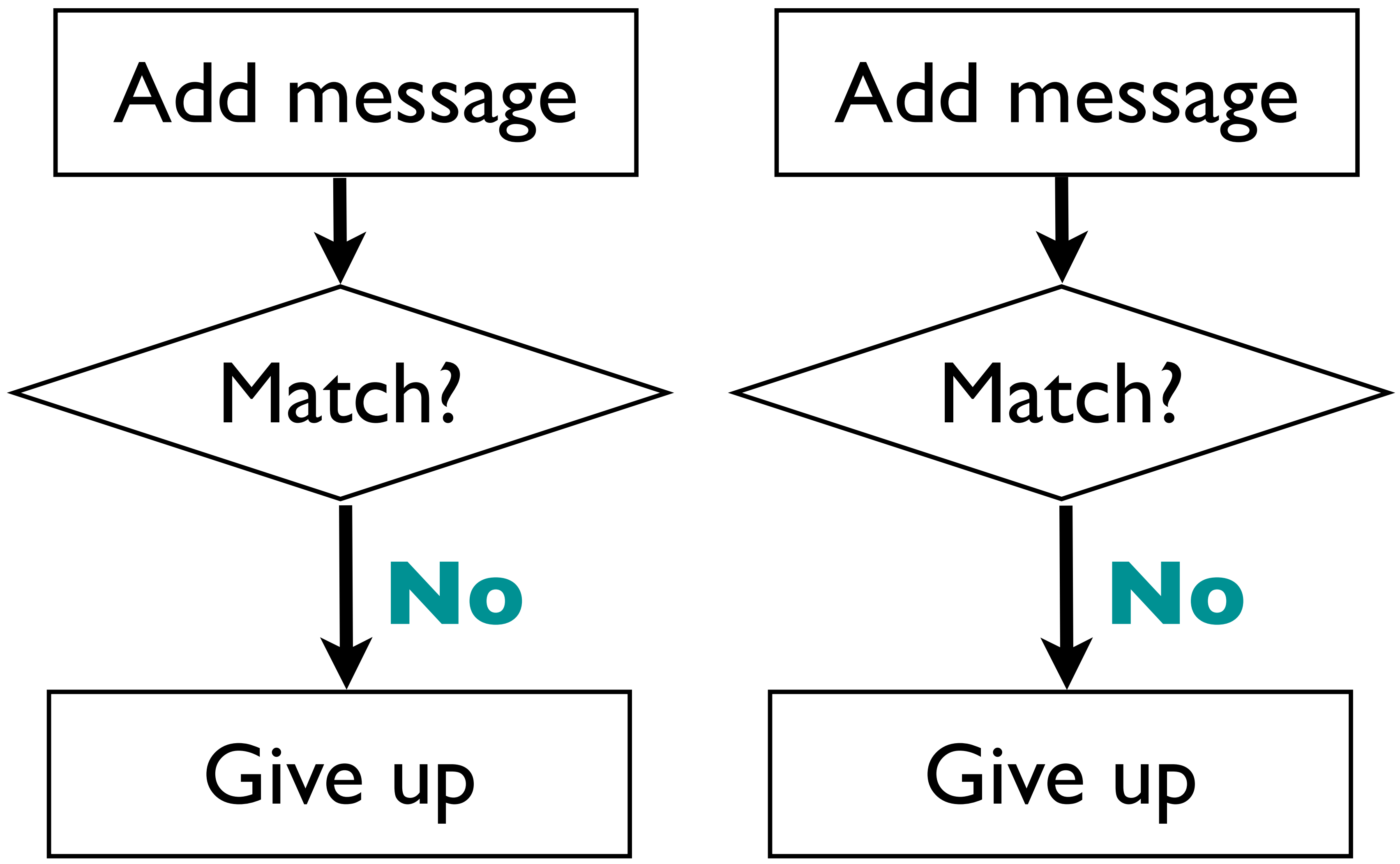
The Protocol



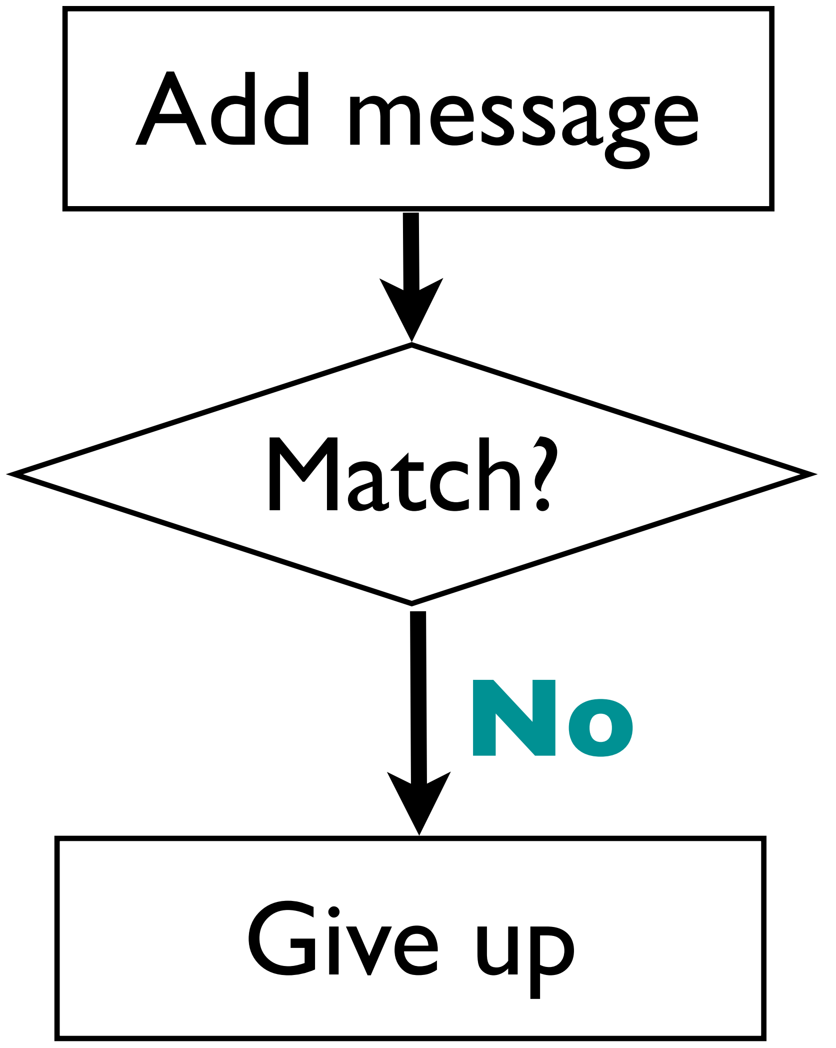
The Protocol

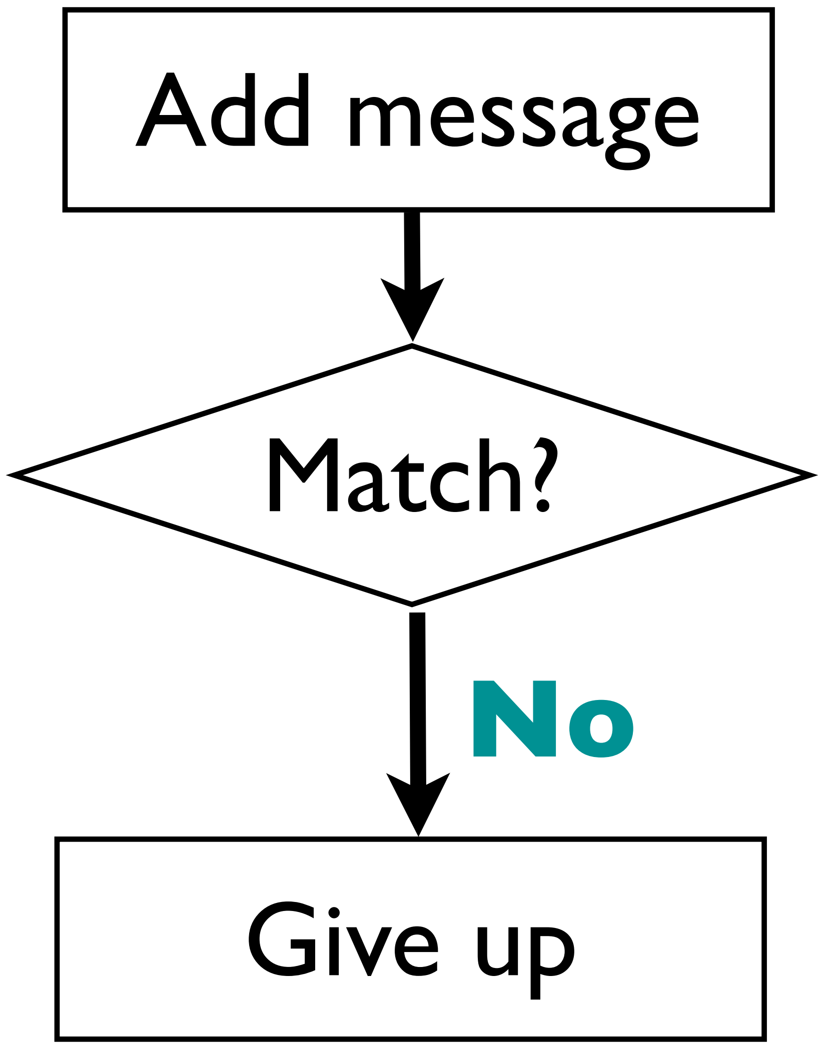


The Protocol



The Protocol





The Paper

- “Lazy” message adding
- Stealing
- Counter channels

The Paper

- "Lazy" message adding
- Stealing
- Counter channels

⇨

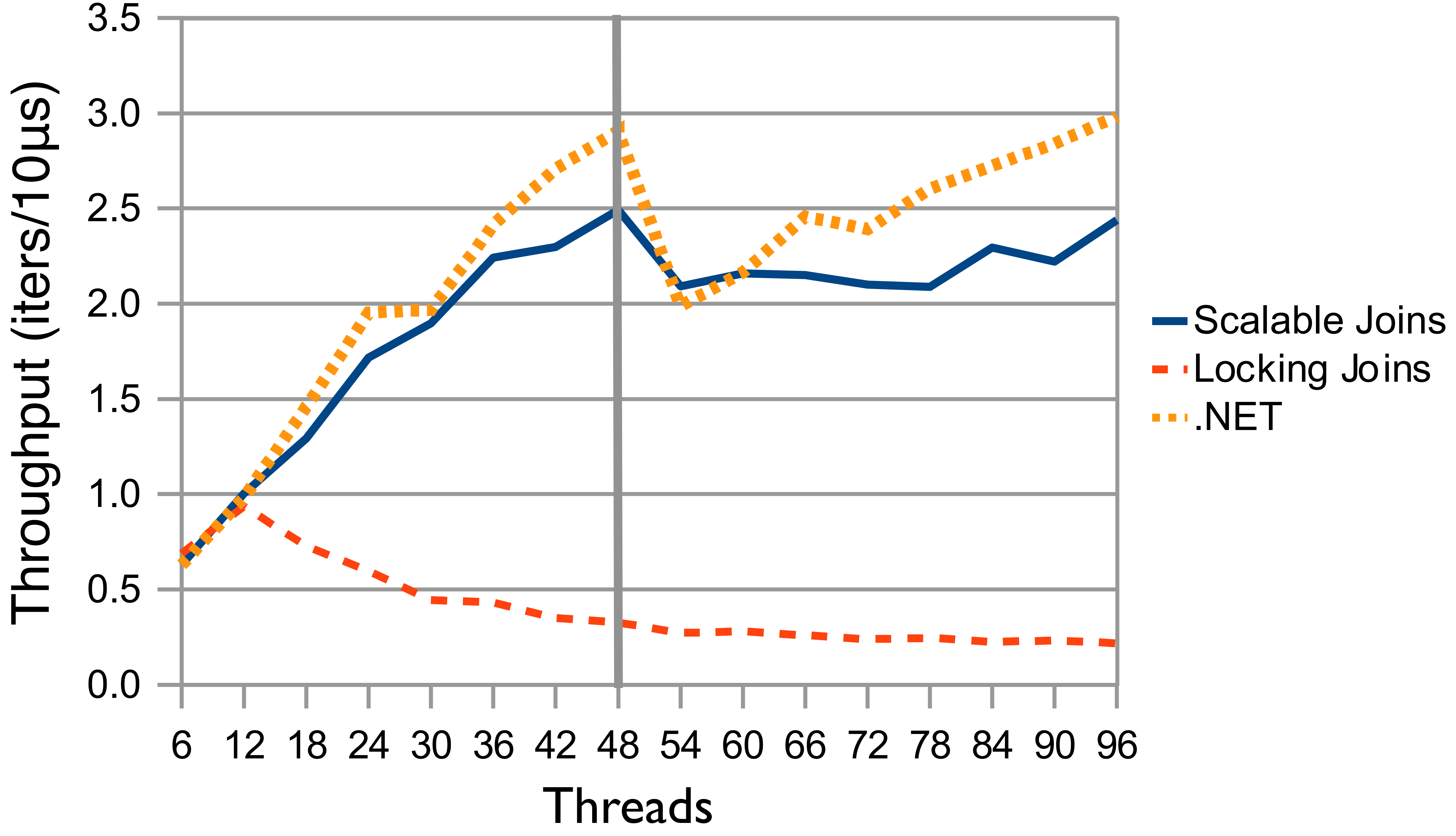
Recover existing algorithms

Benchmarks

48 core AMD machine

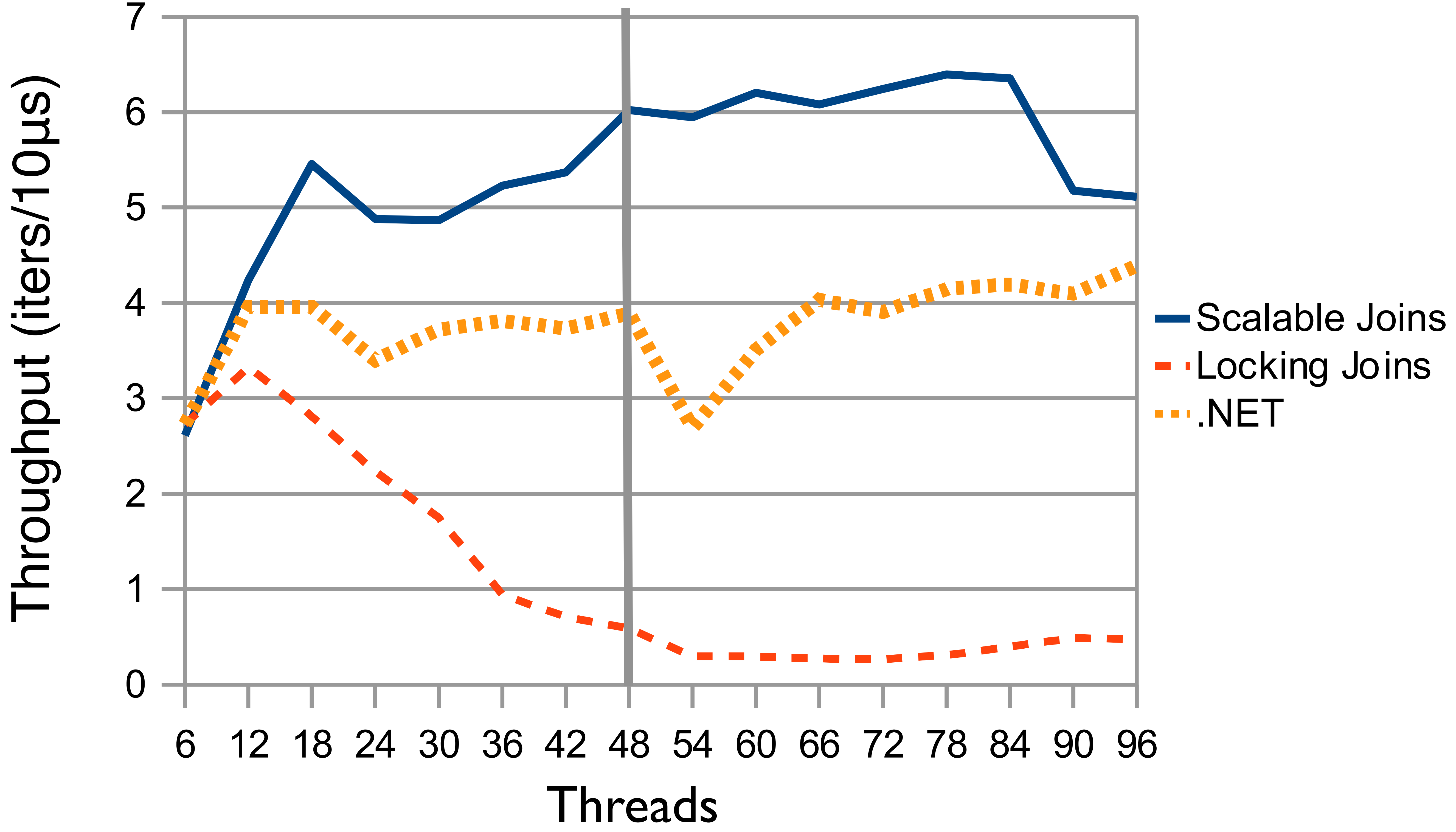
Simulated fine-grained parallel workload

Lock



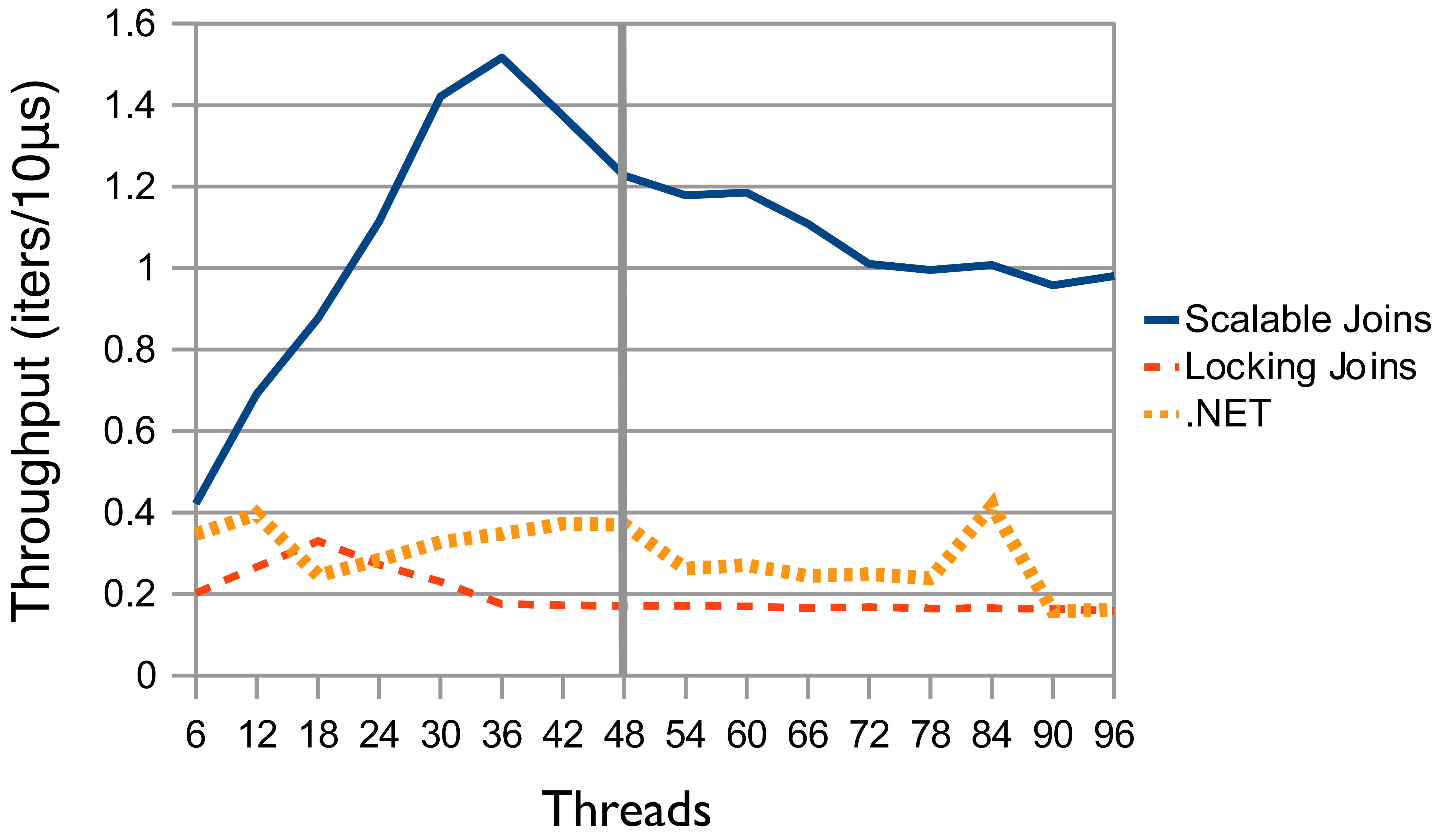
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Throughput (iters/10μs)
6
12
18
24
30
36
42
48
54
60
66
72
78
84
90
96
Threads
Scalable Joins
Locking Joins
.NET

Semaphore



Throughput (iters/10µs)
0
1
2
3
4
5
6
7
6
12
18
24
30
36
42
48
54
60
66
72
78
84
90
96
Threads
Scalable Joins
Locking Joins
.NET

Producer/Consumer



This work

Use ***join patterns*** for synchronization:

[Fournet & Gonthier]

Expressive

Write synchronization primitives declaratively and concisely

Scalable

Competitive with industrial libraries; can recover existing algorithms

The Takeaway

Have your cake and eat it, too

Just one algorithm to implement, endlessly extensible by the user, competitive with custom solutions

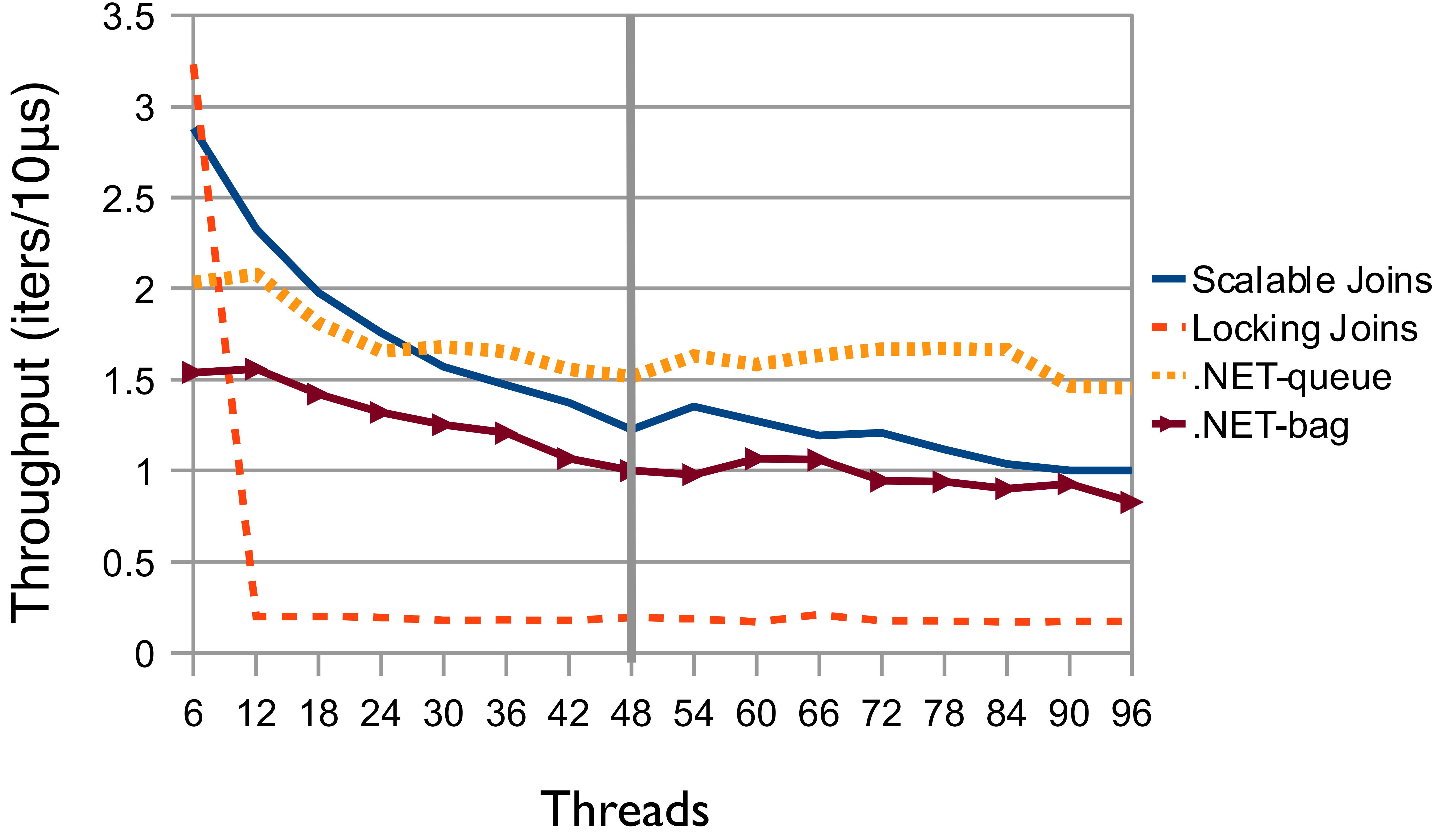
The Takeaway

Have your cake and eat it, too

Just one algorithm to implement, endlessly extensible by the user, competitive with custom solutions

Thank you

Producer/Consumer (no work)


Throughput (iters/10µs)
Threads
Scalable Joins
Locking Joins
.NET-queue
.NET-bag