SoFTeX
Software Technology Group

Technische Universität
Kaiserslautern

Verification of Actor Systems Needs Specification Techniques for Strong Causality and Hierarchical Reasoning

Arnd Poetzsch-Heffter, Ilham W. Kurnia, Christoph Feller

FoVeOOS 2011
Turin, Italy

November 11, 2011

Overview

1. Actor system recap
2. Position
3. Verification Sketch
4. Conclusion

Actor System Recap

- Communication: asynchronous message passing
- Internal state encapsulated
- Concurrent: each executes on its own
- May create more actors
- Assumptions:
  - Unbounded message queue
  - FIFO scheduling, with guards
  - Single-threaded actors, focusing on reactive behavior
  - Message body execution terminates

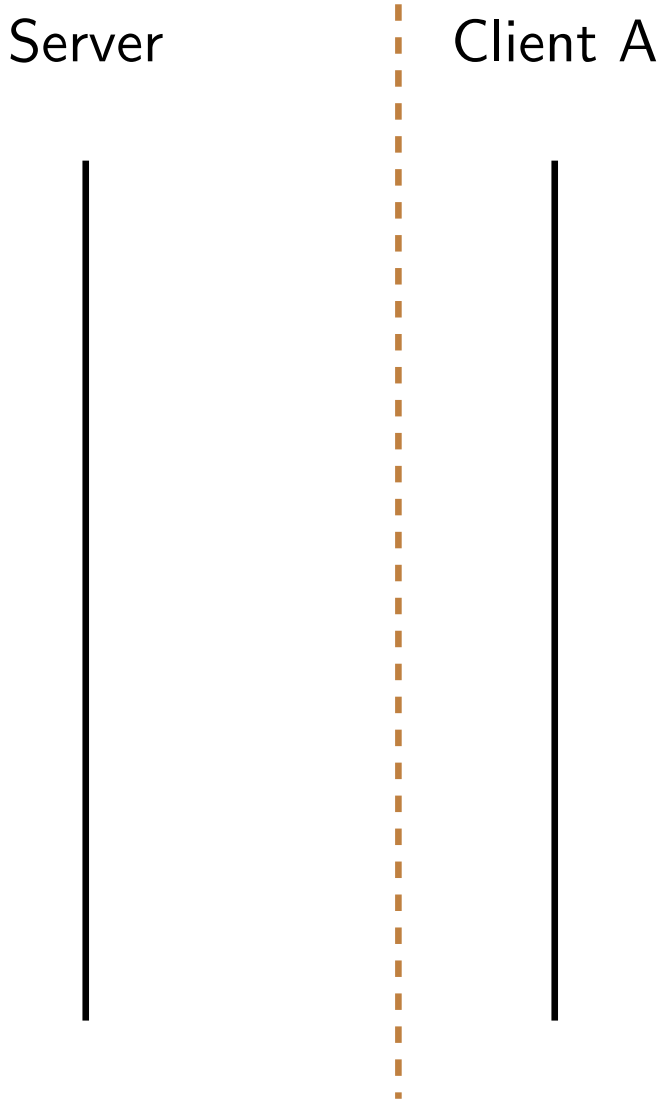
Position

To verify actor systems, one needs specification techniques which can handle

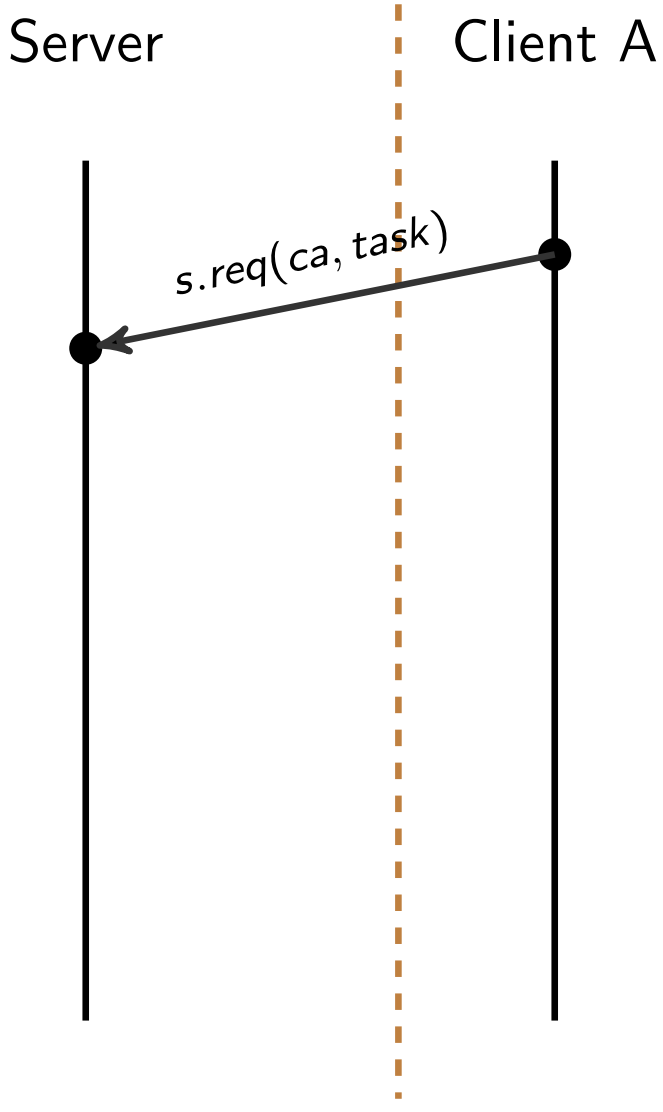
- Strong causality
- Hierarchical reasoning

A specification technique that works for single actors and scales to groups consisting of many actors

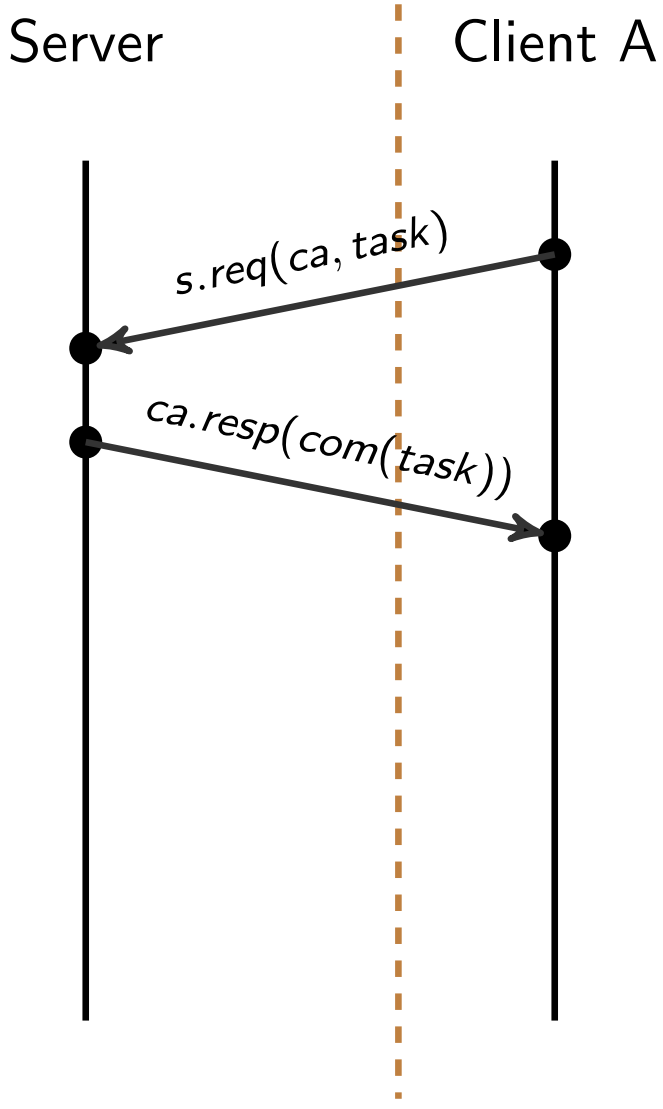
Client-Server Interaction



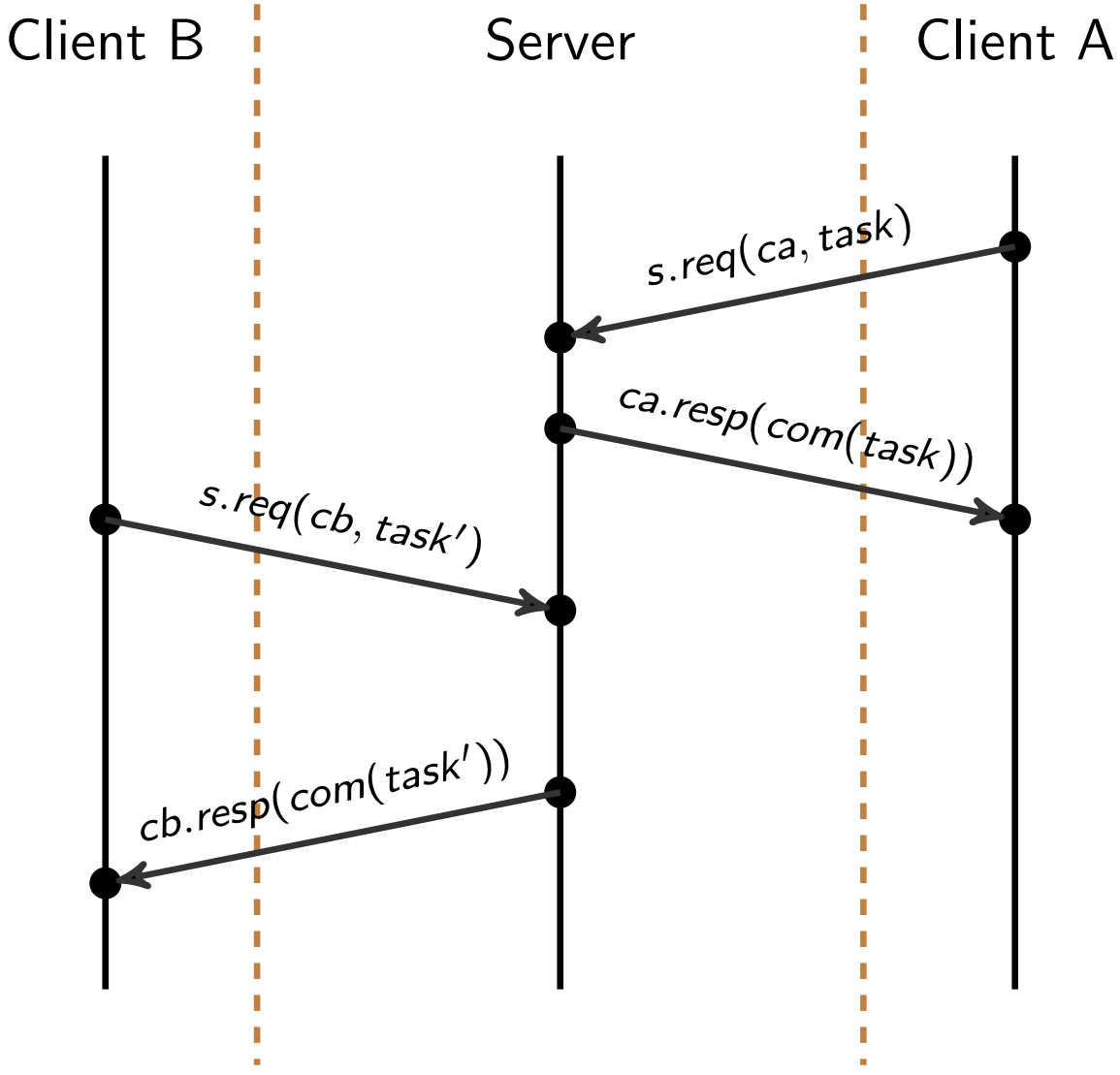
Client-Server Interaction



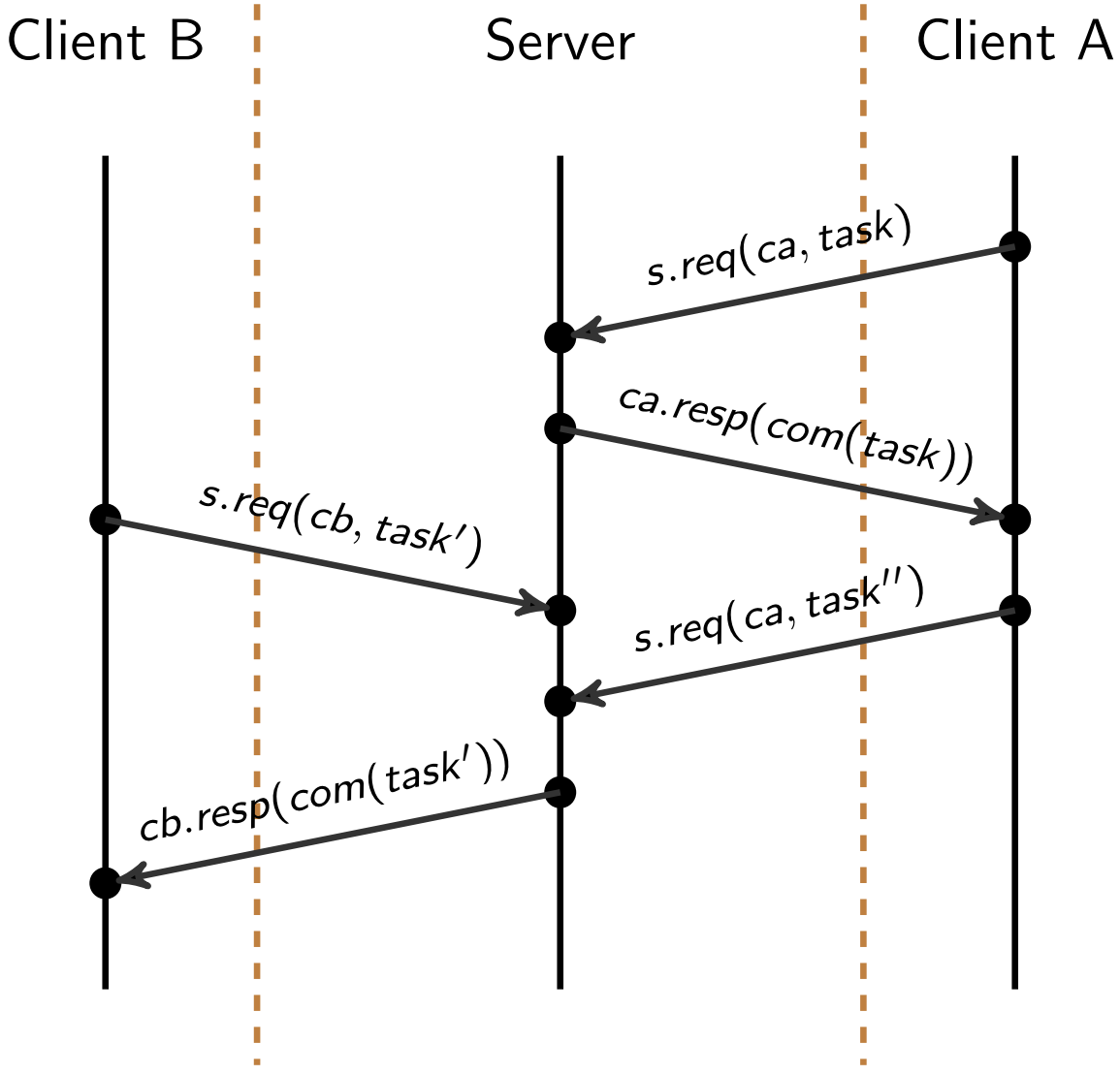
Client-Server Interaction



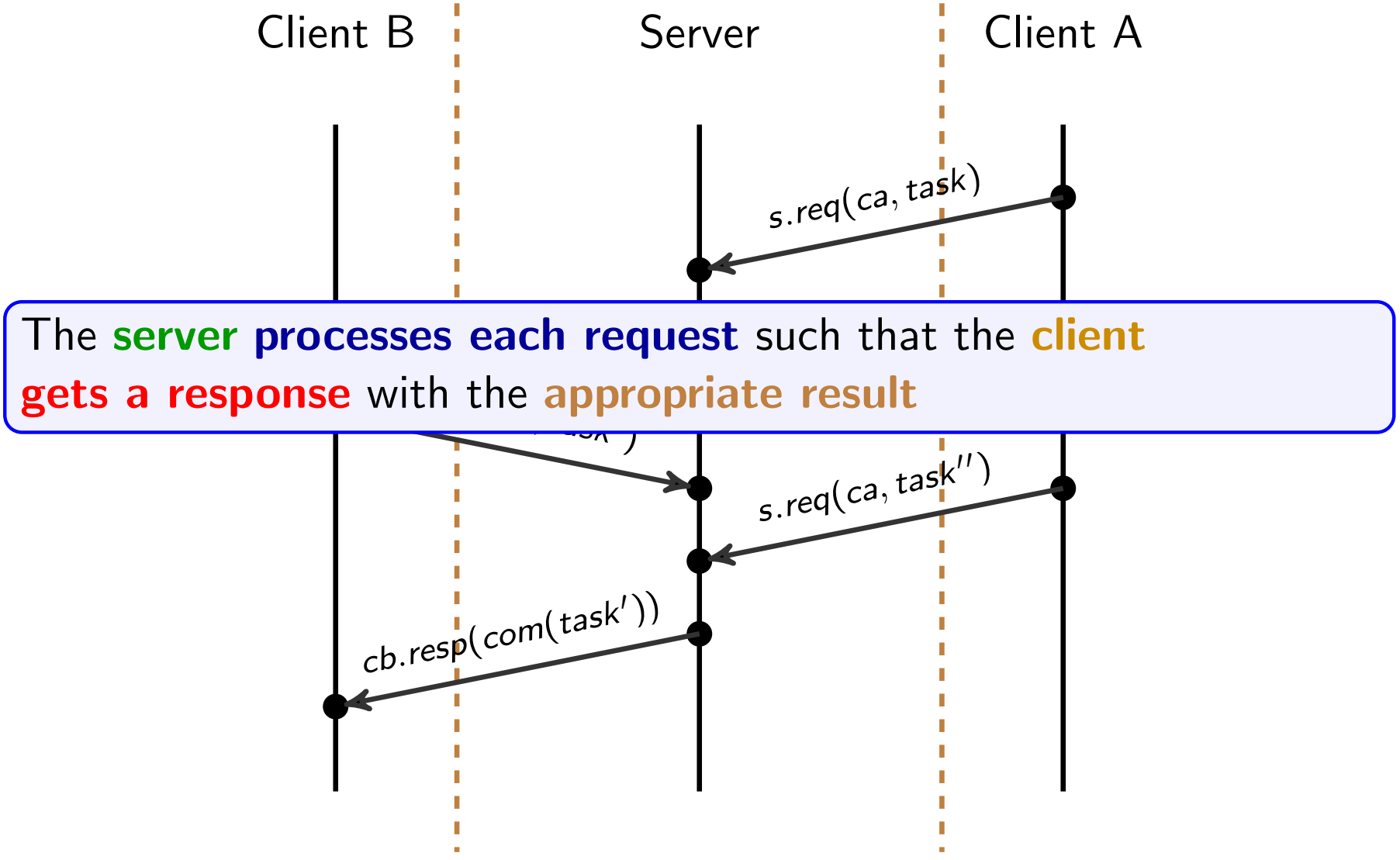
Client-Server Interaction



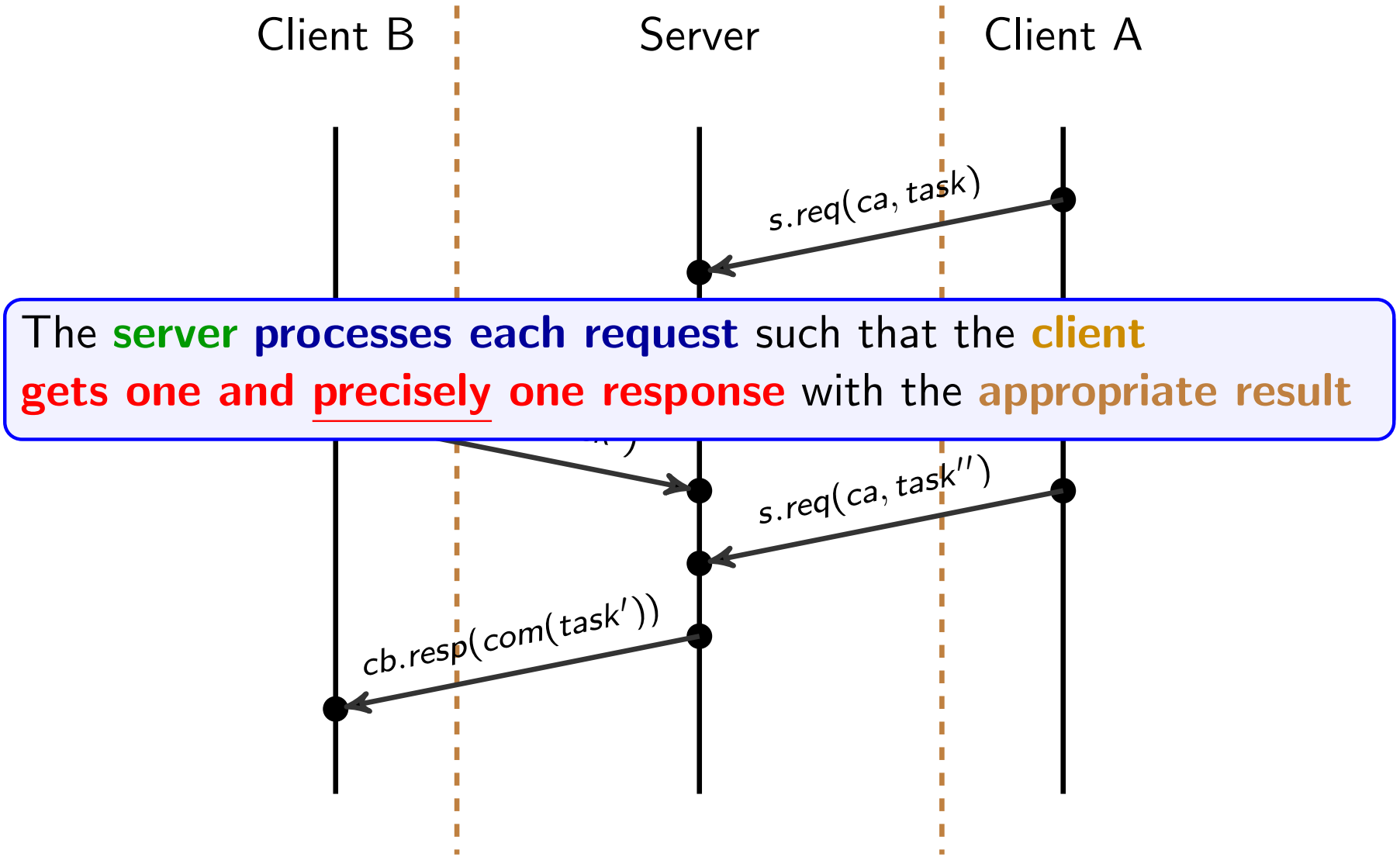
Client-Server Interaction



Client-Server Interaction



Client-Server Interaction

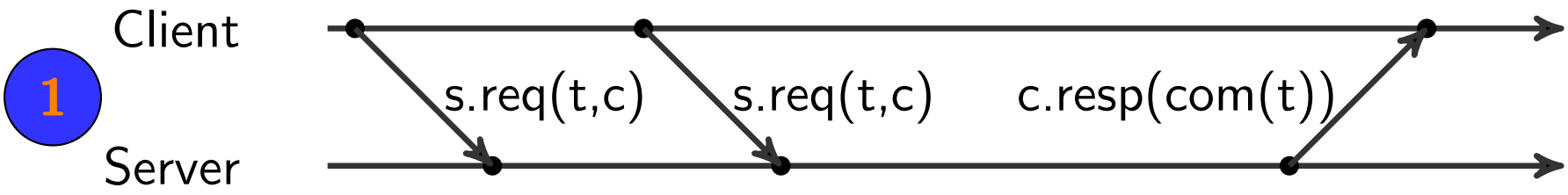


Client-Server Interaction (Operational Semantics)

The **server** **processes each request** such that the **client**
gets one and precisely one response with the **appropriate result**

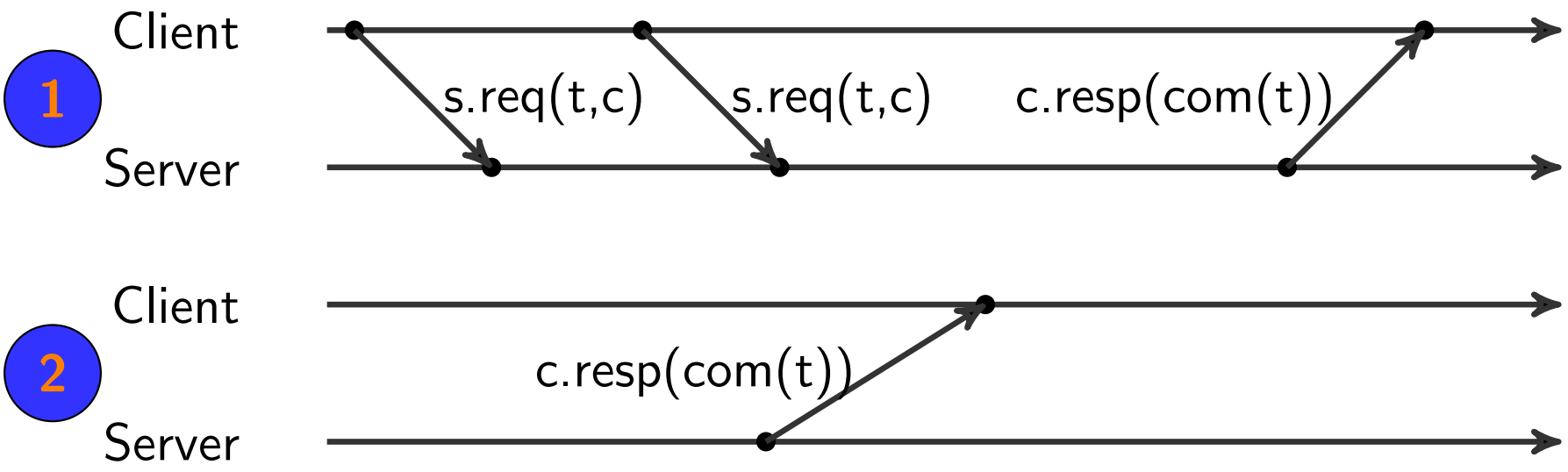
Client-Server Interaction (Operational Semantics)

The **server** **processes each request** such that the **client**
gets one and precisely one response with the **appropriate result**



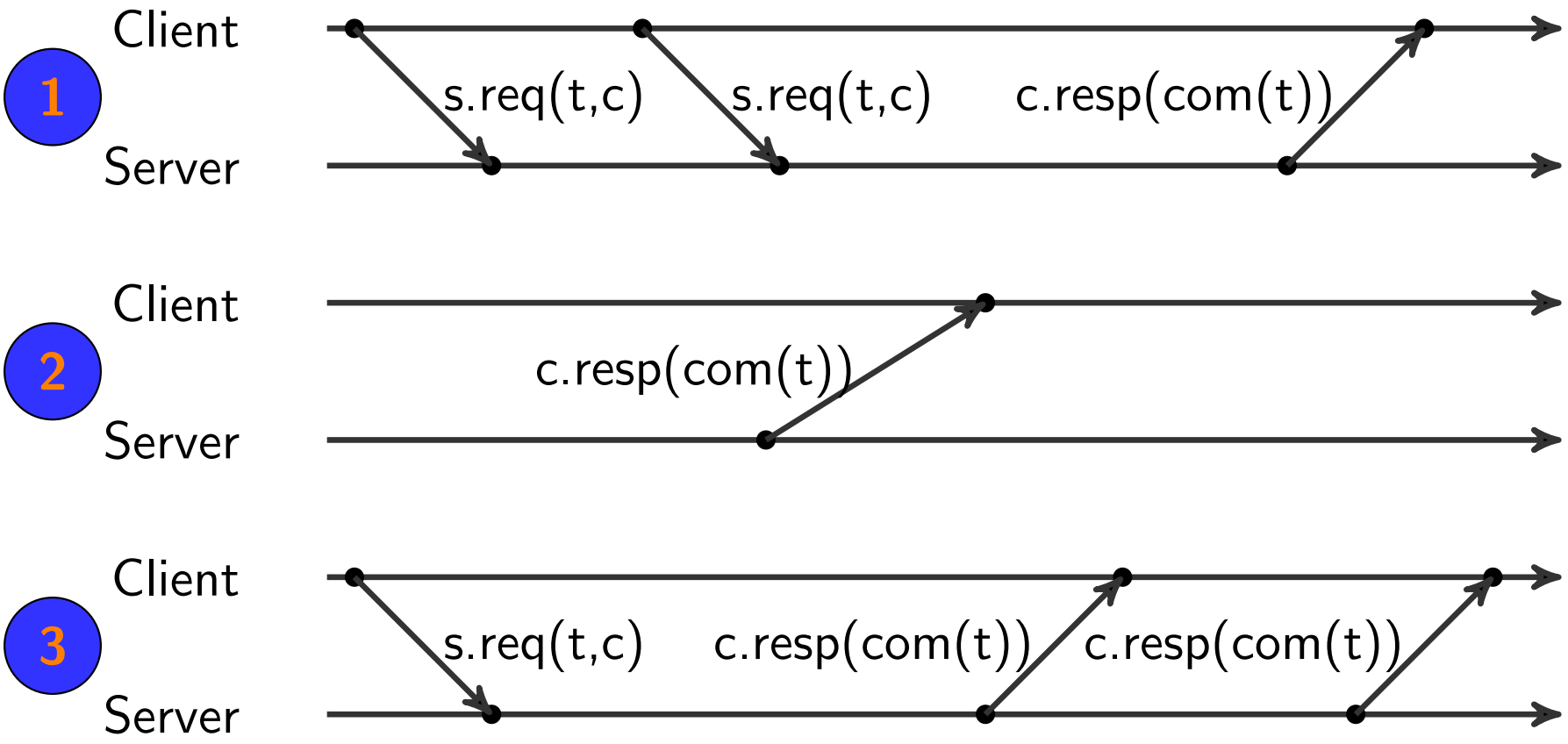
Client-Server Interaction (Operational Semantics)

The **server** **processes each request** such that the **client** **gets one and precisely one response** with the **appropriate result**



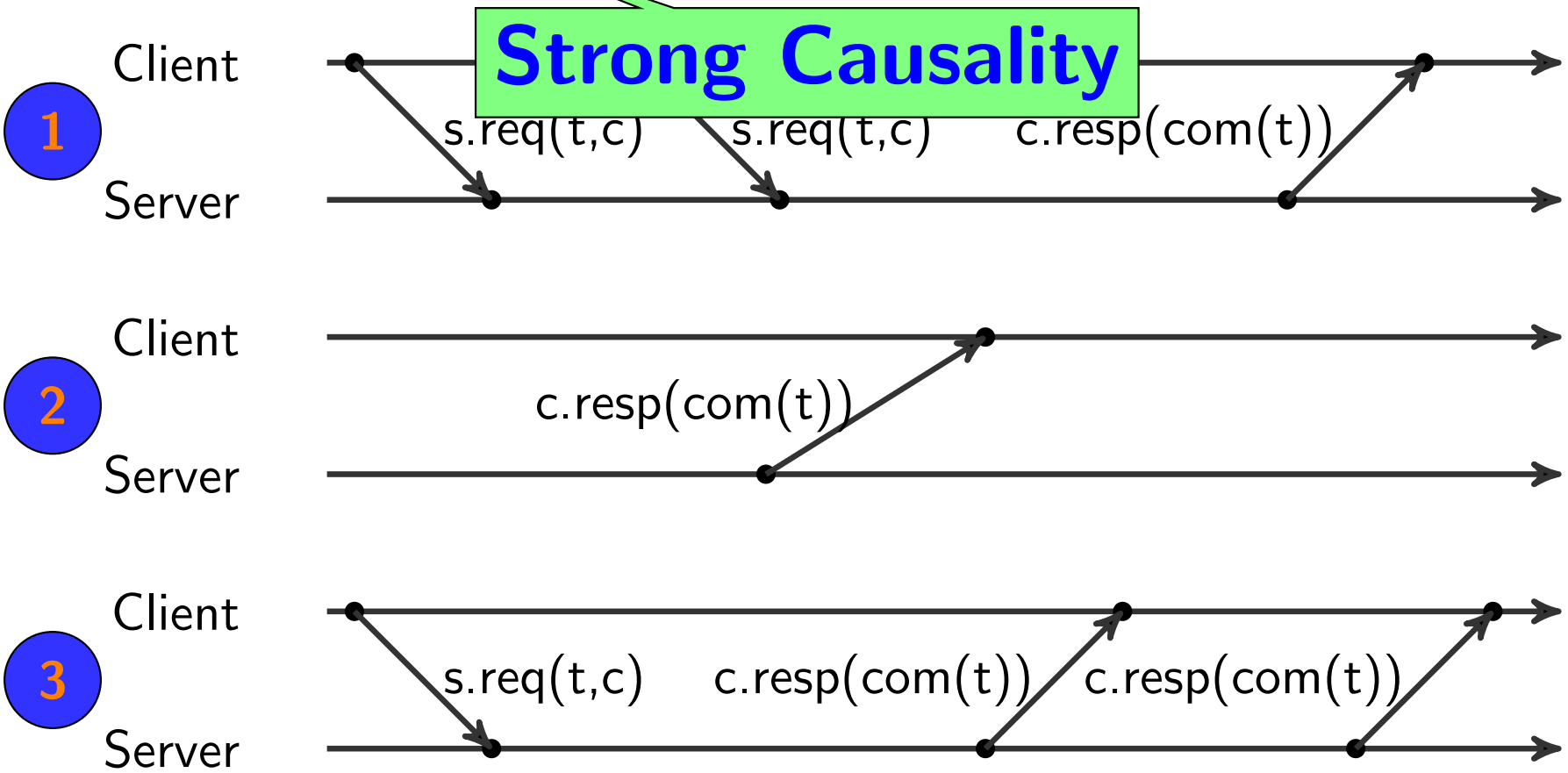
Client-Server Interaction (Operational Semantics)

The **server** **processes each request** such that the **client** **gets one and precisely one response** with the **appropriate result**



Client-Server Interaction (Operational Semantics)

The **server** **processes each request** such that the **client** **gets one and precisely one response** with the **appropriate result**



Specification Techniques (1)

LTL [Pnueli, 1977]

$$\forall t, c \ \cdot \Box\left(s.req\left(t, c\right) \rightarrow \Diamond c.resp\left(com\left(t\right)\right)\right)$$

Specification Techniques (1)

LTL [Pnueli, 1977]

$$\forall t, c \cdot \Box\left(s.req\left(t, c\right) \rightarrow \Diamond c.resp\left(com\left(t\right)\right)\right)$$

Counting LTL [Laroussinie et al., 2010]

$$\forall t, c \cdot \Box_{\#s.req(t,c)-\#c.resp(com(t))\geq 0} \top$$

Specification Techniques (1)

LTL [Pnueli, 1977]

$\forall t, c \cdot \Box\left(s.req\left(t, c\right) \rightarrow \Diamond c.resp\left(com\left(t\right)\right)\right)$

Counting LTL [Laroussinie et al., 2010]

$\forall t, c \cdot \Box_{\#s.req(t,c)-\#c.resp(com(t))\geq 0} \top$ Only **safety** property

Specification Techniques (2)

History-Based [e.g., Johnsen and Owe, 2002]

```
∀ h · ok(h)

ok([]) = true
ok(h⊢m) = ok(h), WHERE m ≠ c.resp(_)
ok(h⊢c.resp(com(t))) = okcnt(h⊢c.resp(com(t)), c.resp(com(t)), 0) ∧
                       ok(h)

okcnt([], _, n) = (n ≥ 0)
okcnt(h⊢m, c.resp(com(t)), n) = okcnt(h, c.resp(com(t)), n)
  WHERE m ≠ c.resp(com(t)) ∧ m ≠ s.req(t, c)
okcnt(h⊢s.req(t, c),c.resp(com(t)),n)=okcnt(h,c.resp(com(t)),n+1)
okcnt(h⊢c.resp(com(t)),c.resp(com(t)),n)=okcnt(h,c.resp(com(t)),n-1)
```

Specification Techniques (3)

History-Based with Ready Set [e.g., Owe, 1998]

$s/h \leftarrow s.req(t,c) \implies s/h \vdash s.req(t,c) \nleftarrow c.resp(comp(t))$

Specification Techniques (3)

History-Based with Ready Set [e.g., Owe, 1998]

$s/h \leftarrow s.req(t,c) \implies s/h \vdash s.req(t,c) \nleftarrow c.resp(comp(t))$

Session Types [e.g., Dezani-Ciancaglini et al., 2009]

```
session ClientRequest = begin.!Task.?Value.end
```

Specification Construct

input_message	$\Longrightarrow$	set of	*actor_creation*;
		REGEXP	*output_messages*
	assume	*boolean_expression*	
	assert	*boolean_expression*	

Specification Construct

input_message	$\Longrightarrow$	set of	*actor_creation*;
		REGEXP	*output_messages*
	assume	*boolean_expression*	
	assert	*boolean_expression*	

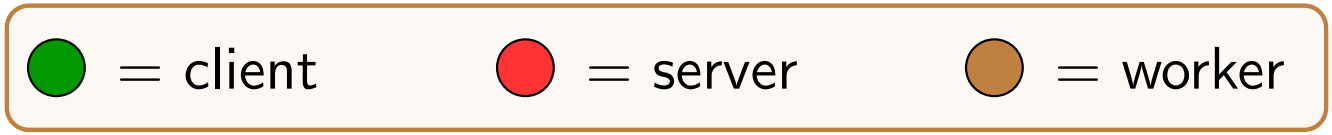
Semantic Basis

- **Events**: (Msg, id)
- **Executions**: alternating sequences of **states** and **events** + **causes**
- **Causes** relation : Event $\times$ Event

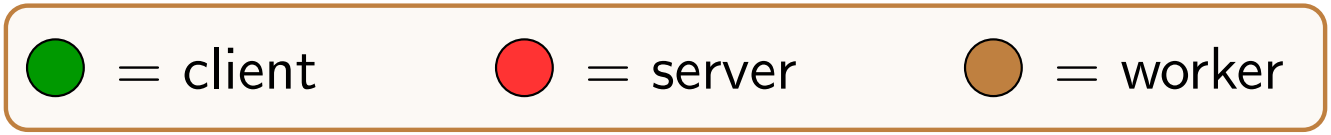
Server Specification

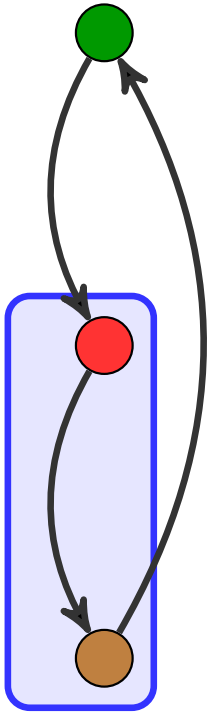
```
group spec AServer {
   this.request(c, t) ==> c.response( compute(t) )
   assumes  c != null
}
```

Server Implementation

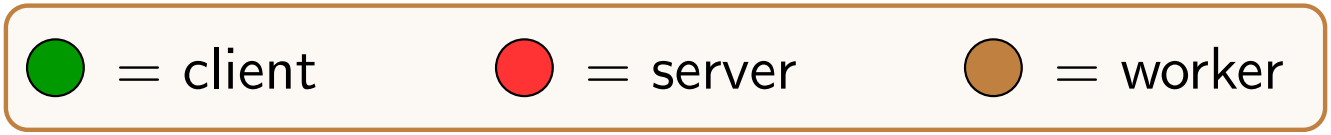


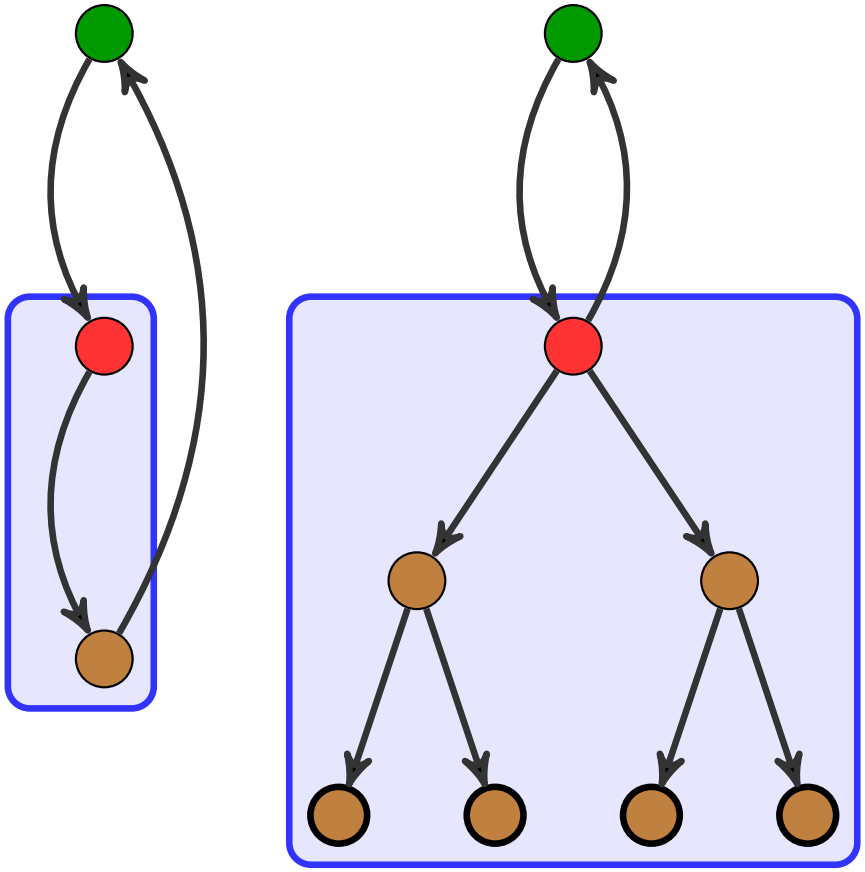
Server Implementation



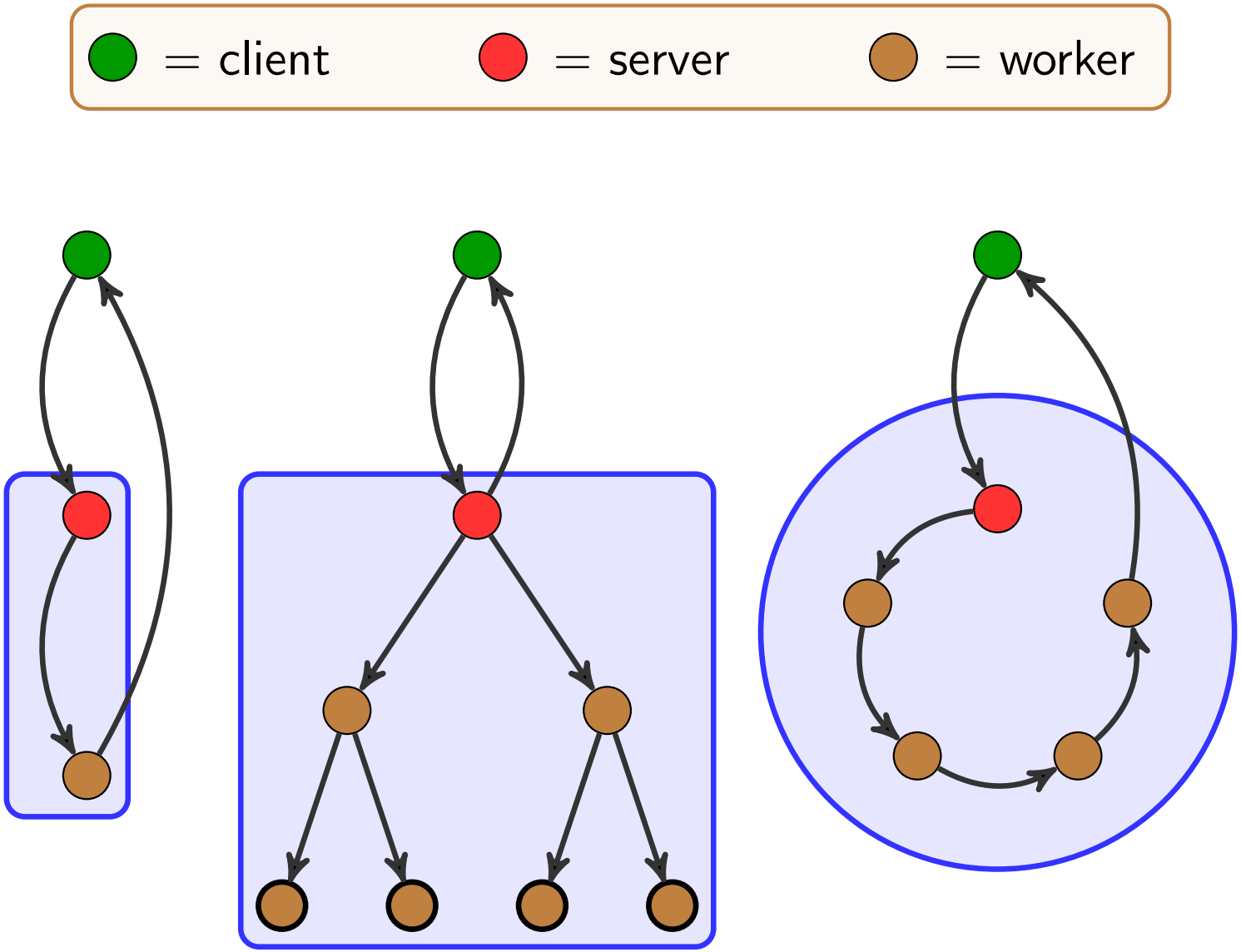


Server Implementation

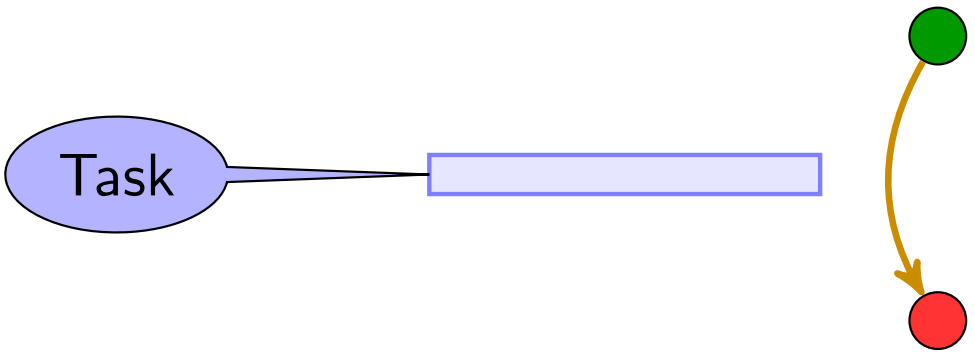




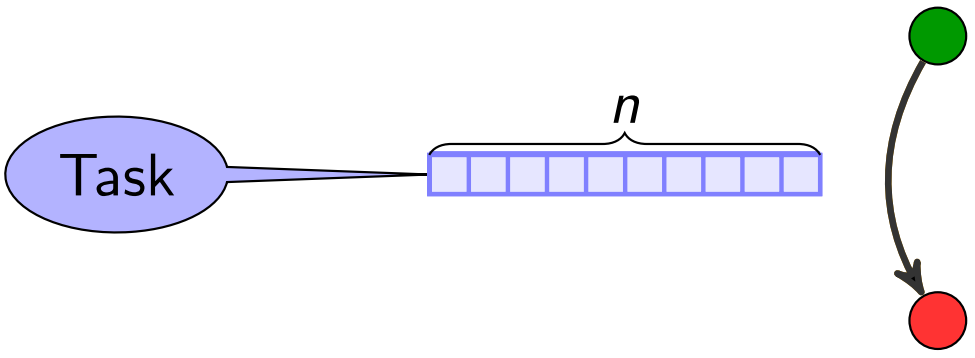
Server Implementation



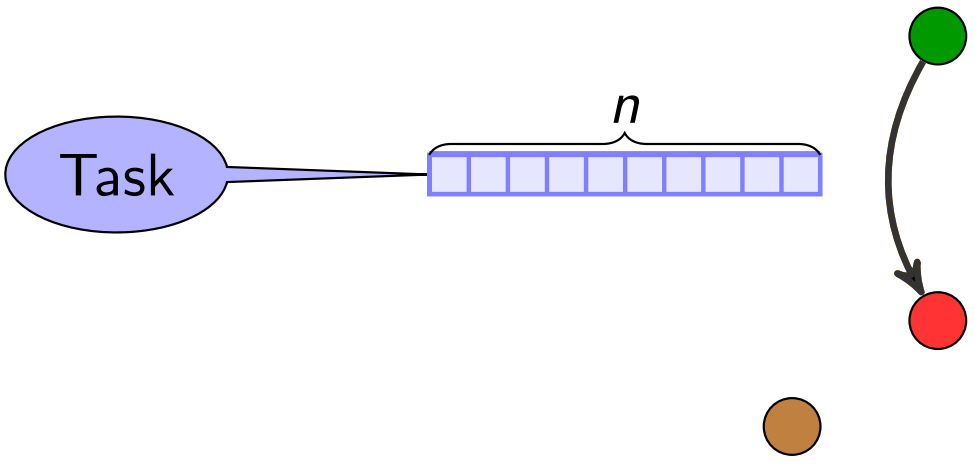
Ring Workers [Arts and Dam, 1999]



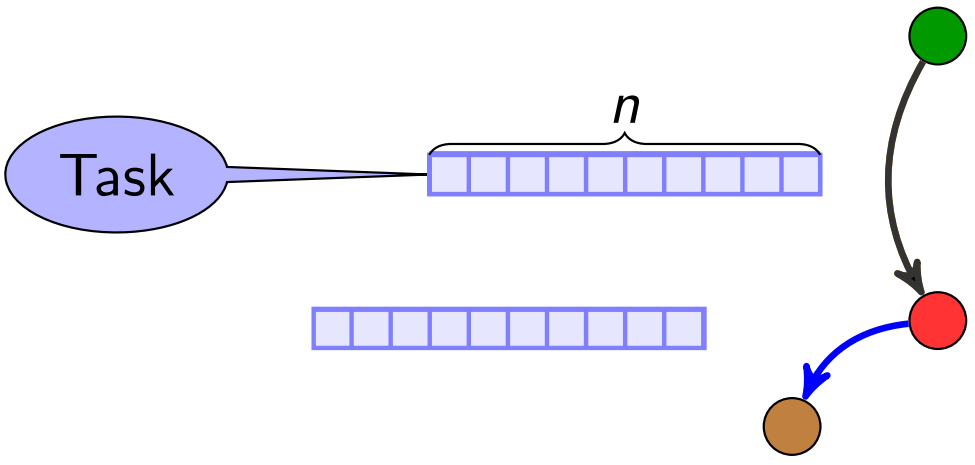
Ring Workers [Arts and Dam, 1999]



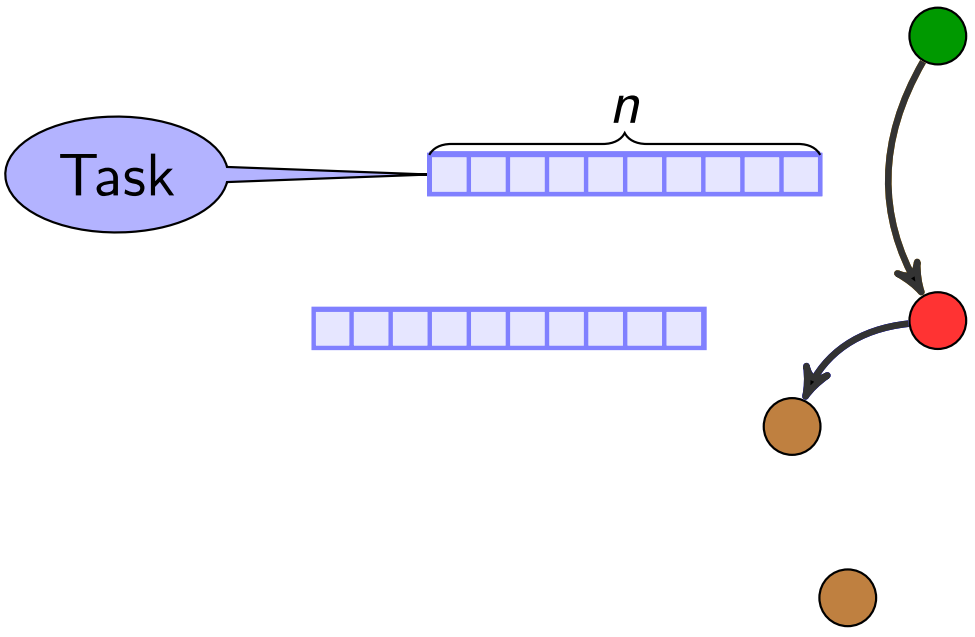
Ring Workers [Arts and Dam, 1999]



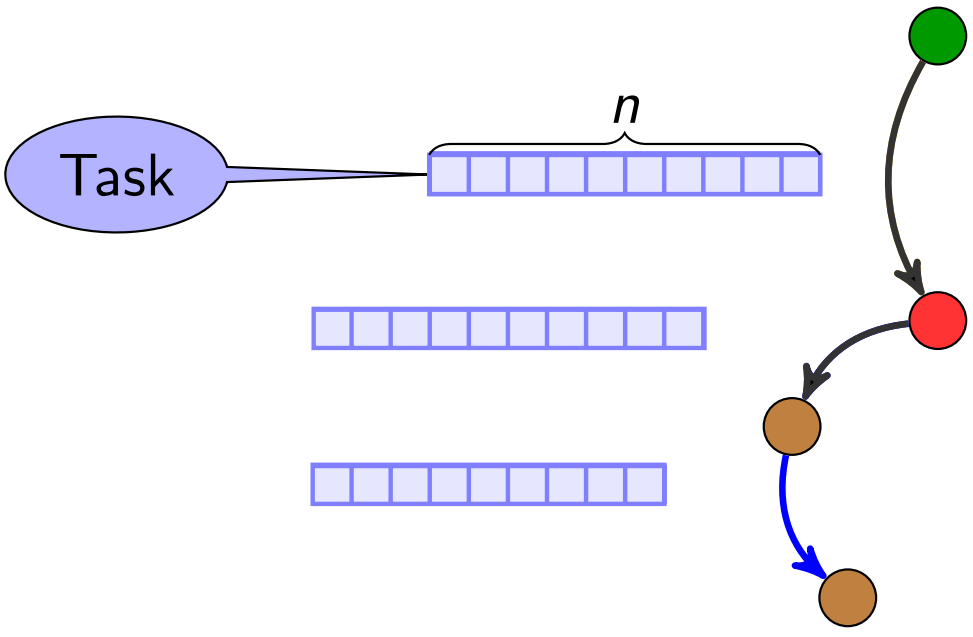
Ring Workers [Arts and Dam, 1999]



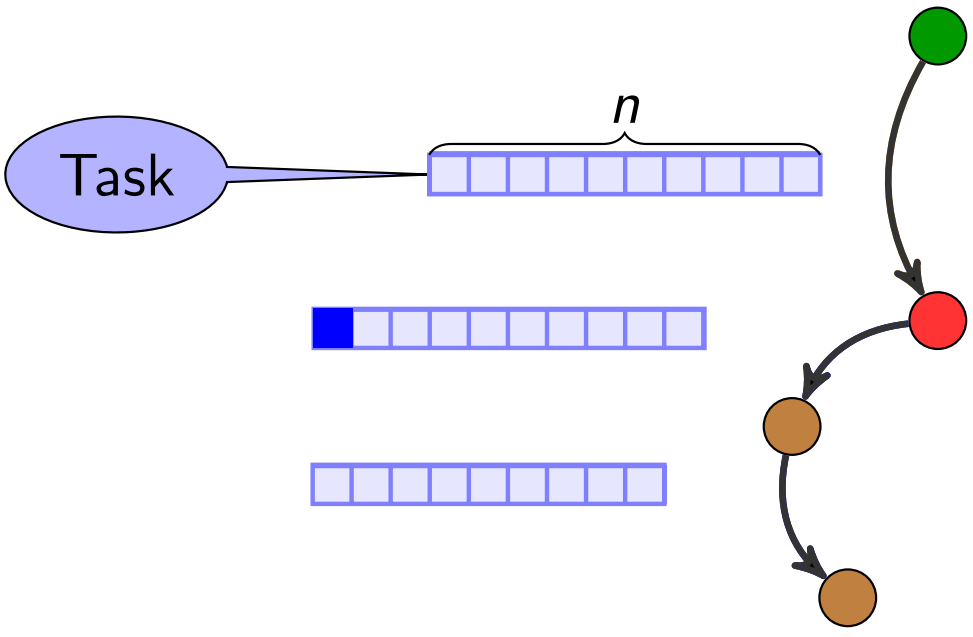
Ring Workers [Arts and Dam, 1999]



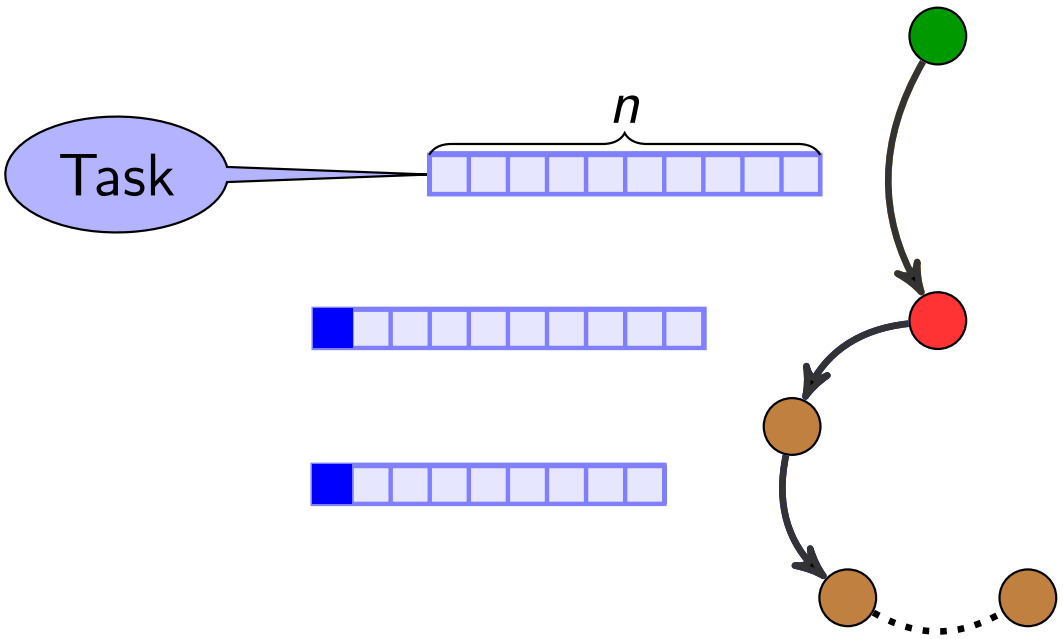
Ring Workers [Arts and Dam, 1999]



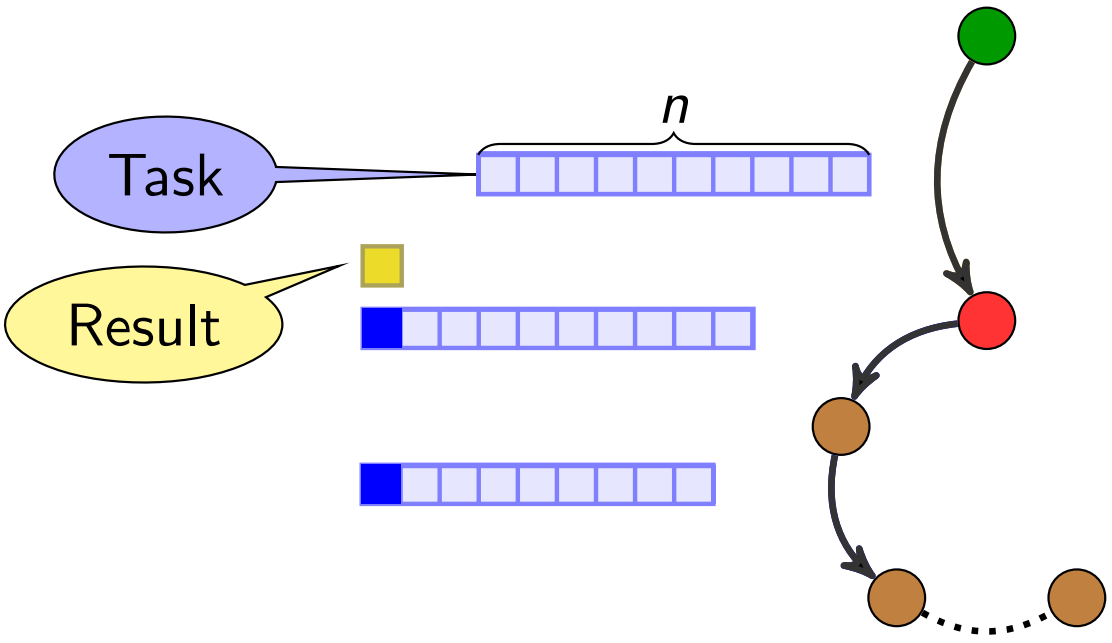
Ring Workers [Arts and Dam, 1999]



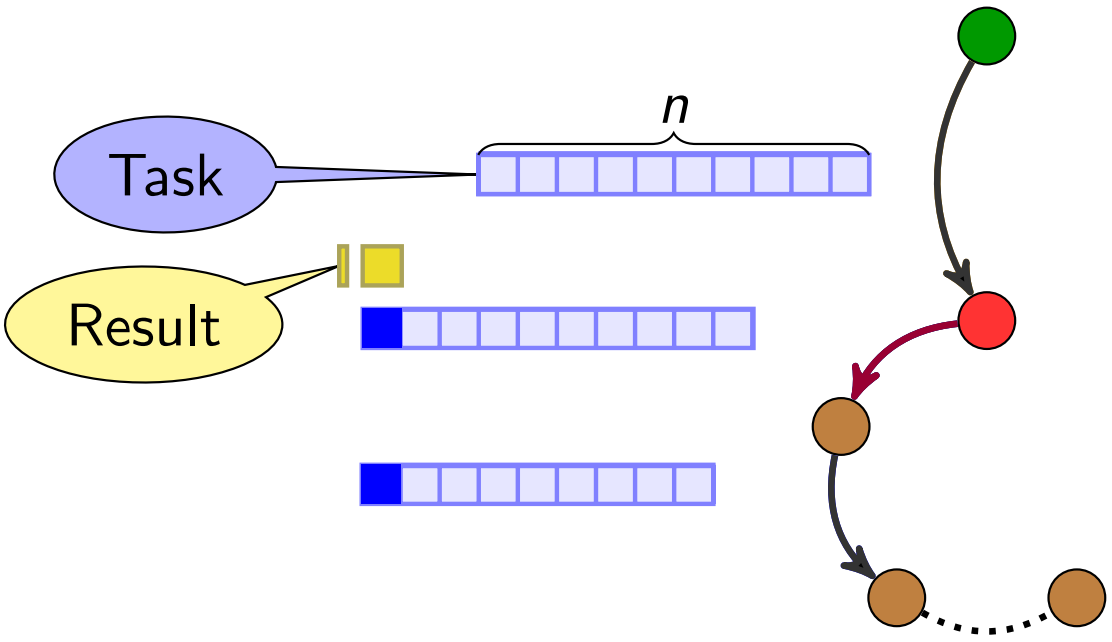
Ring Workers [Arts and Dam, 1999]



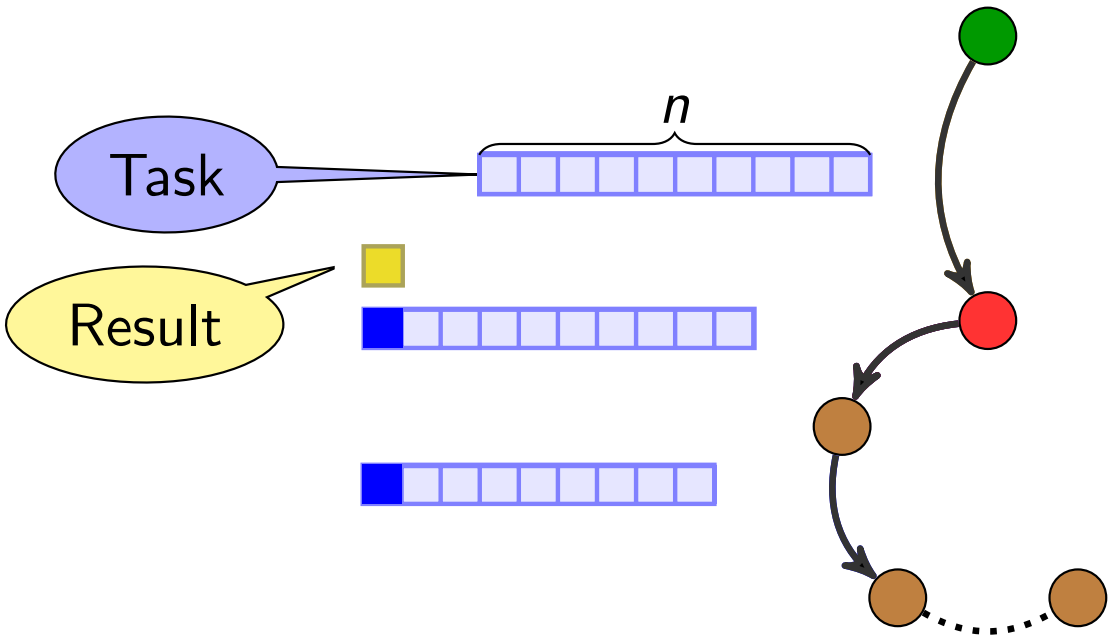
Ring Workers [Arts and Dam, 1999]



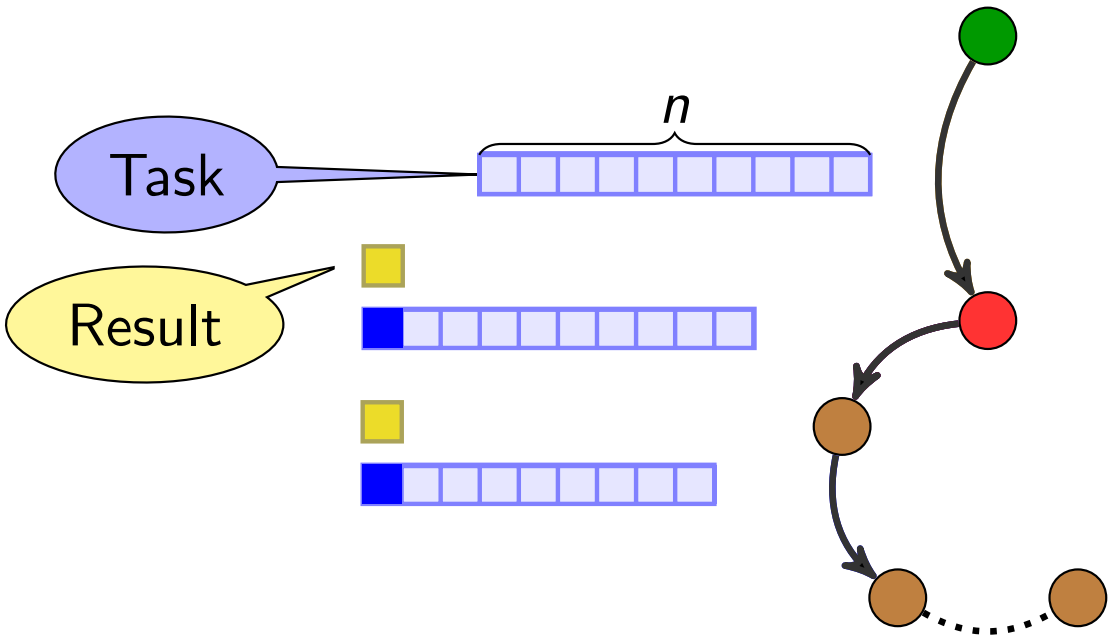
Ring Workers [Arts and Dam, 1999]



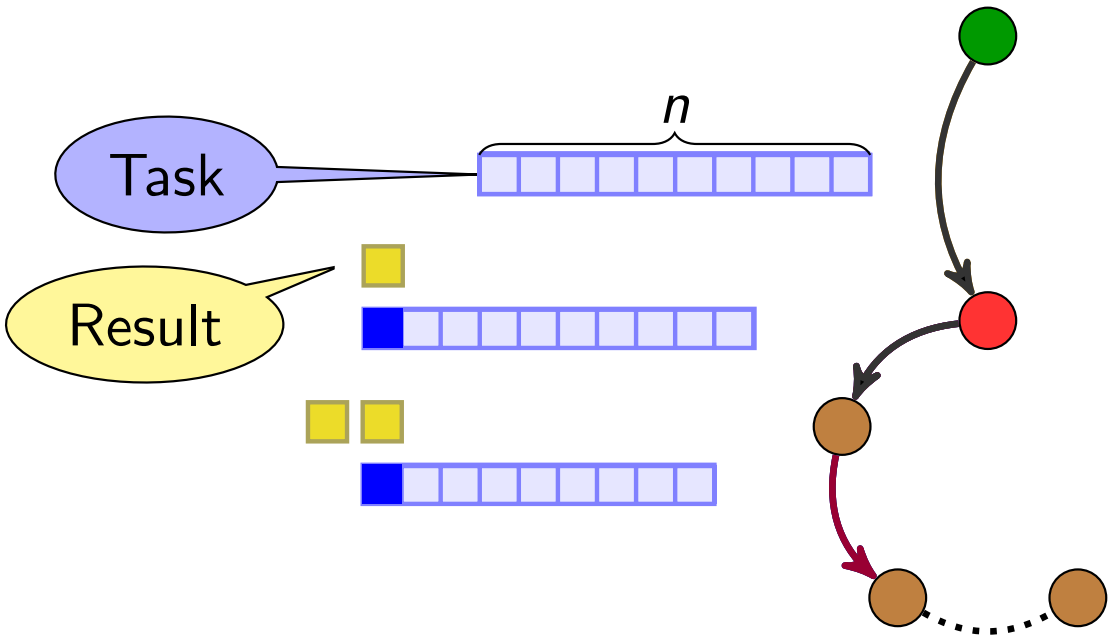
Ring Workers [Arts and Dam, 1999]



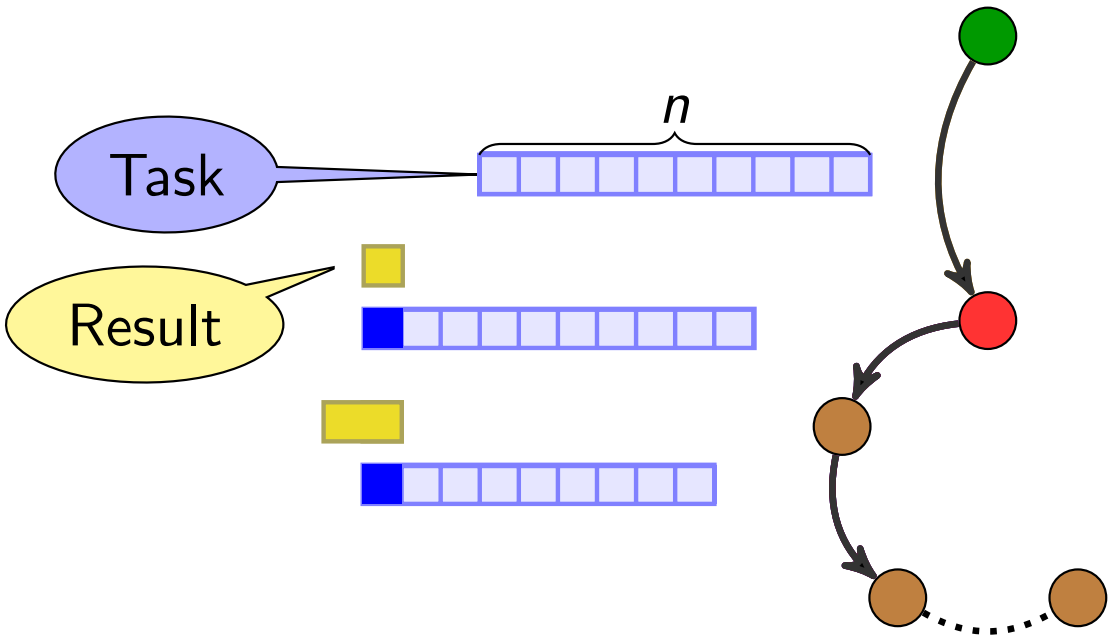
Ring Workers [Arts and Dam, 1999]



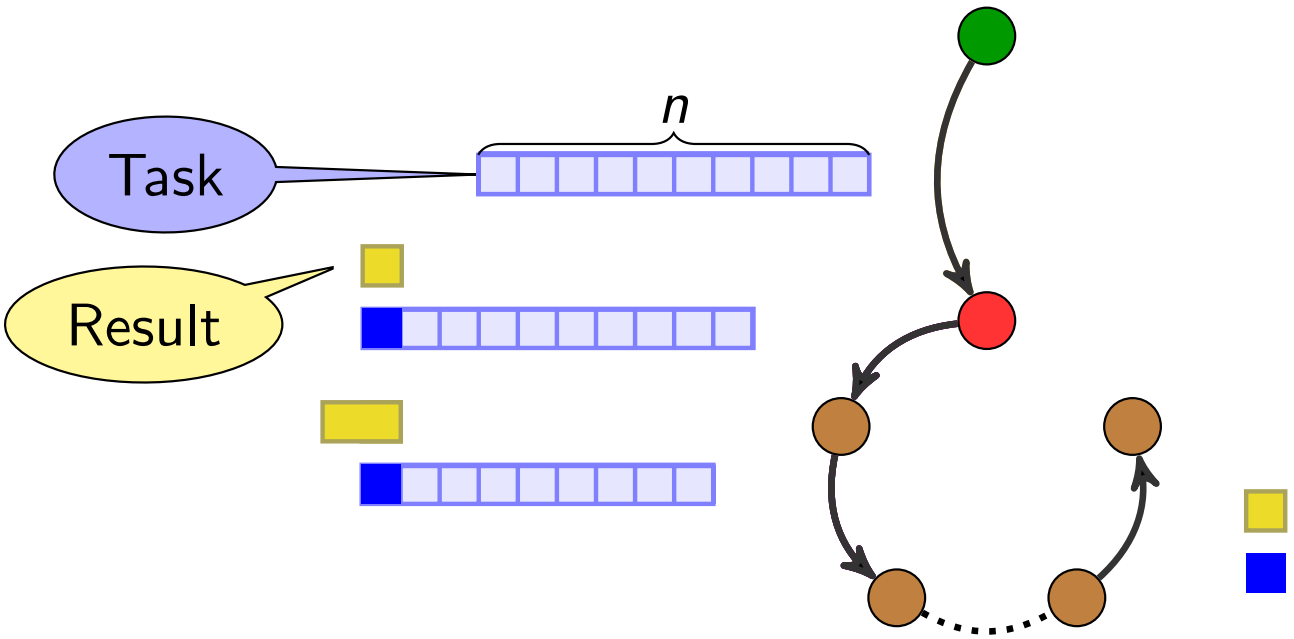
Ring Workers [Arts and Dam, 1999]



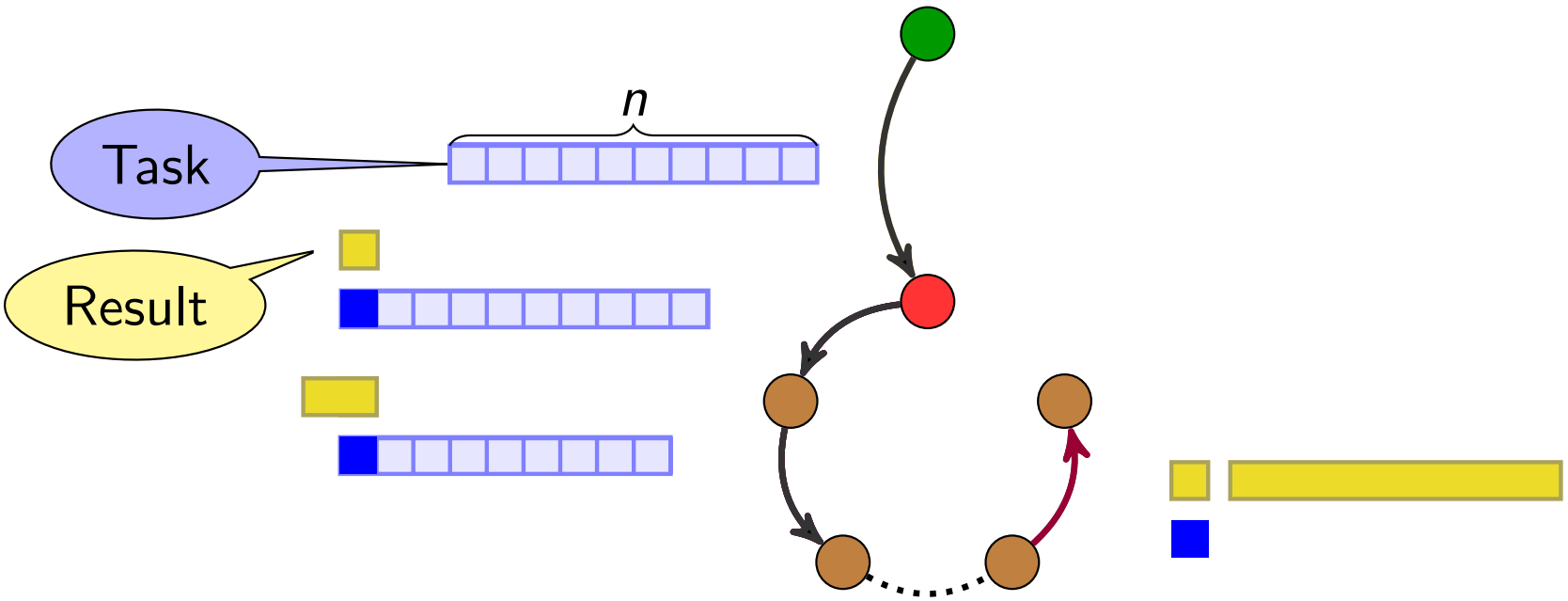
Ring Workers [Arts and Dam, 1999]



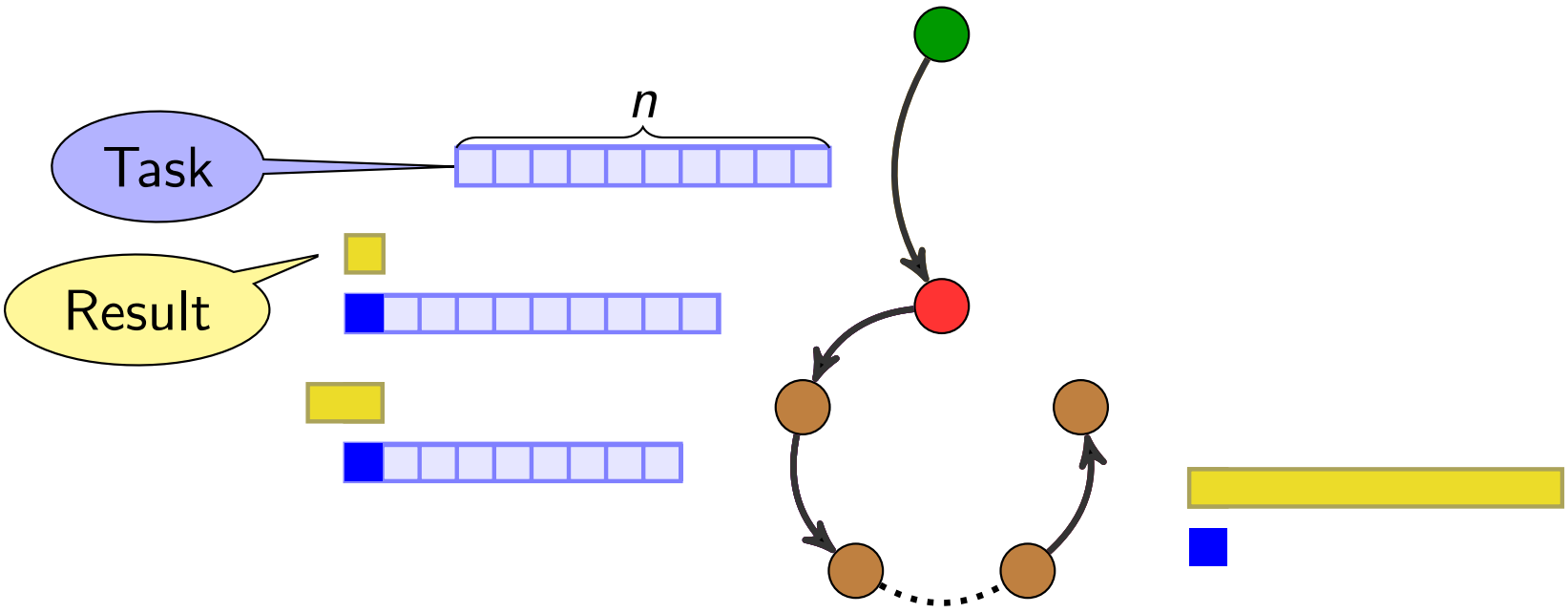
Ring Workers [Arts and Dam, 1999]



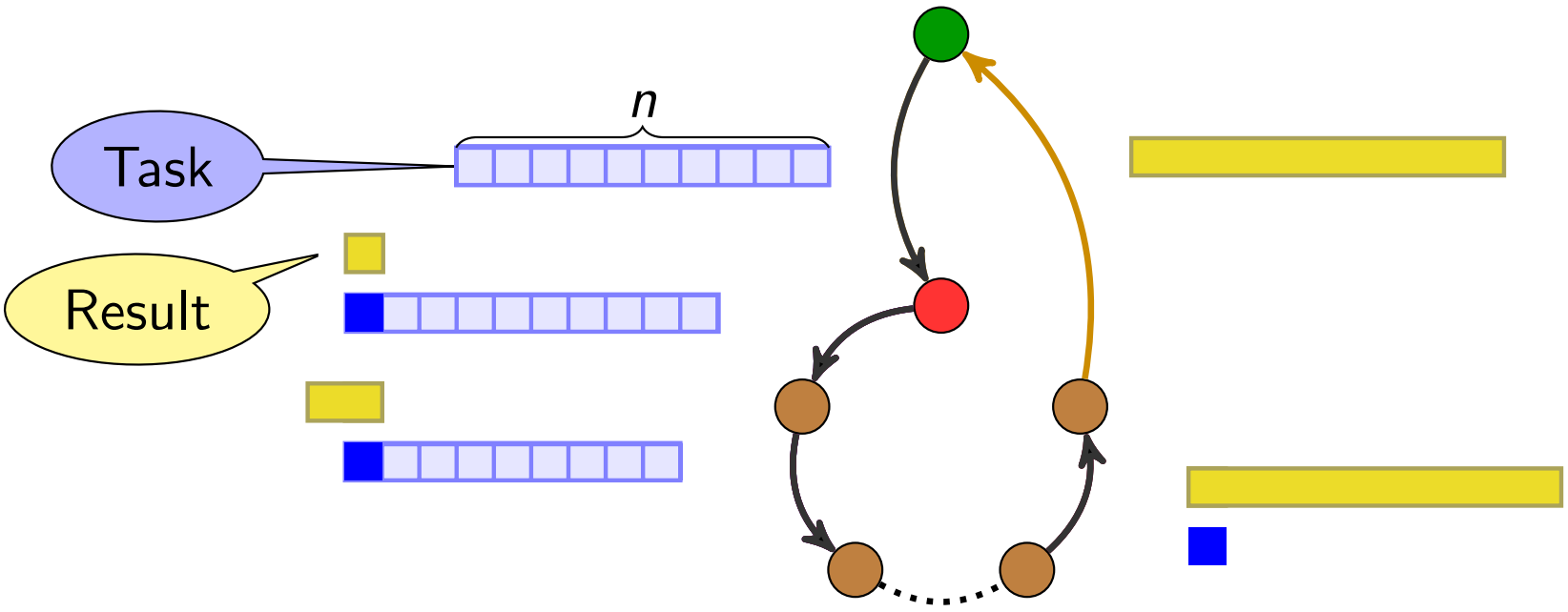
Ring Workers [Arts and Dam, 1999]



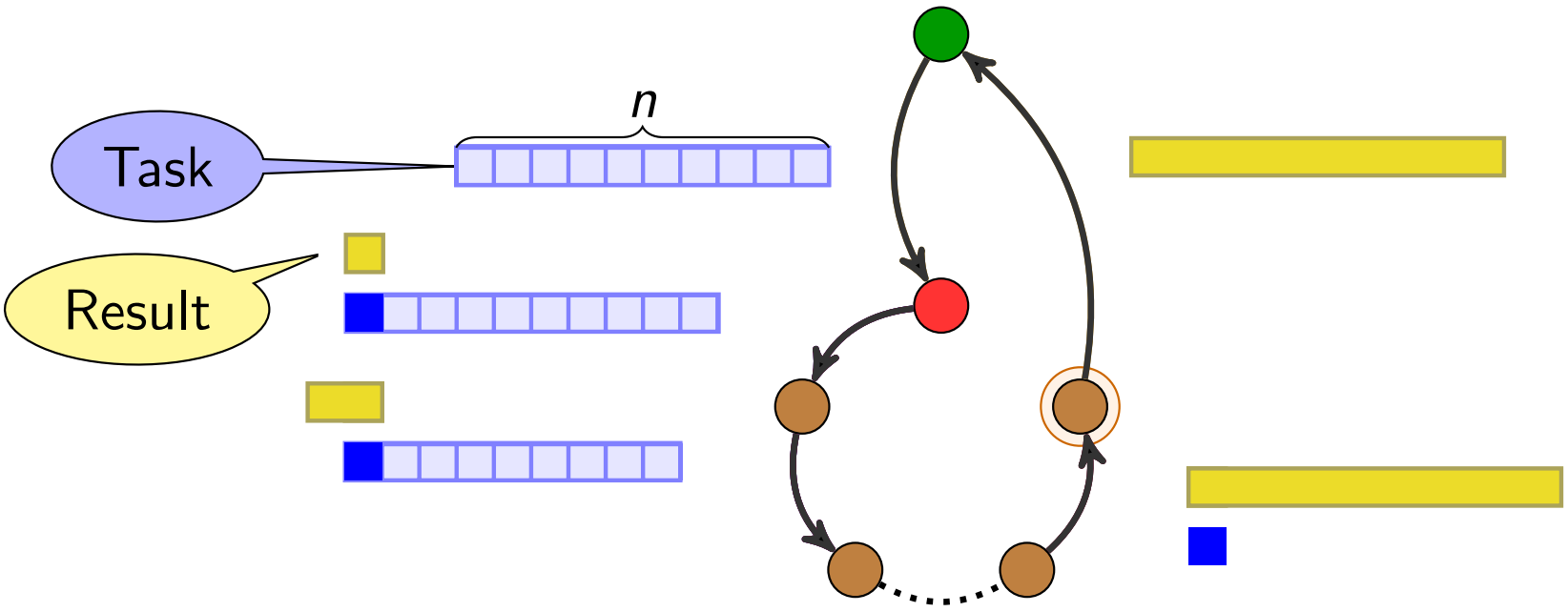
Ring Workers [Arts and Dam, 1999]



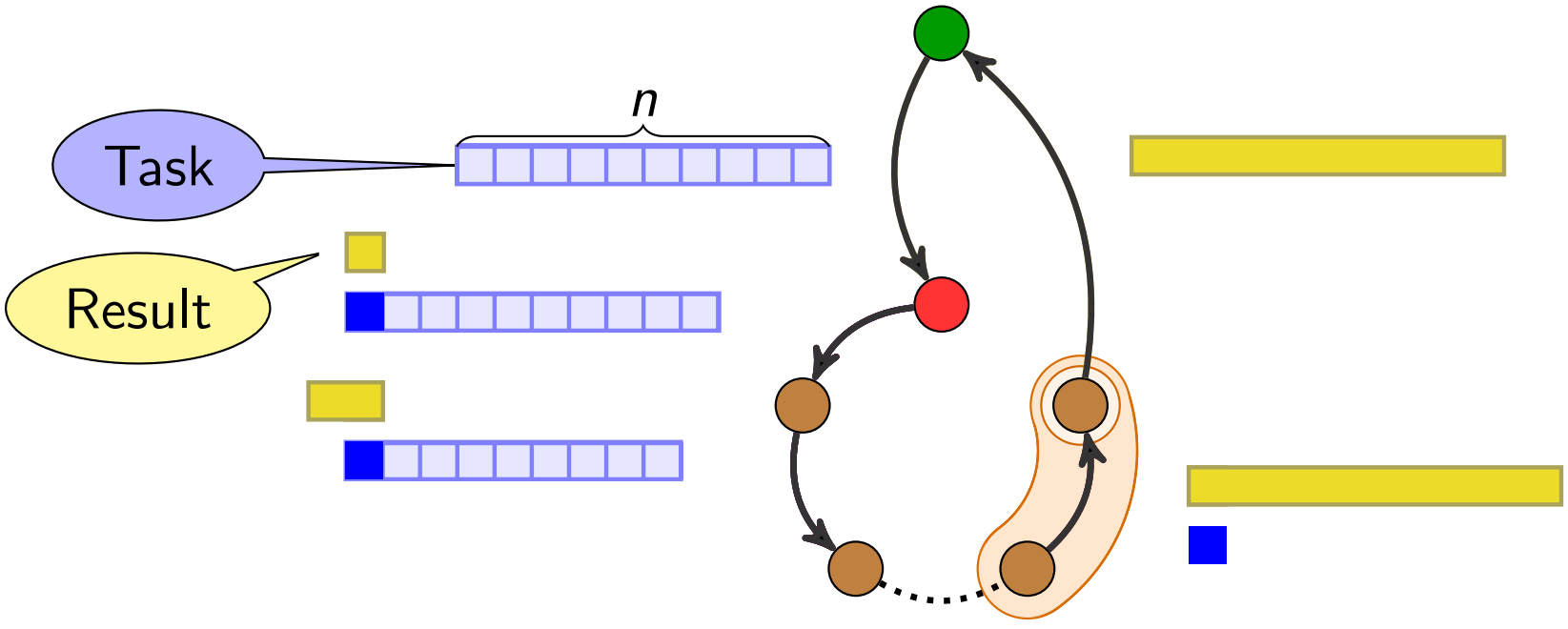
Ring Workers [Arts and Dam, 1999]



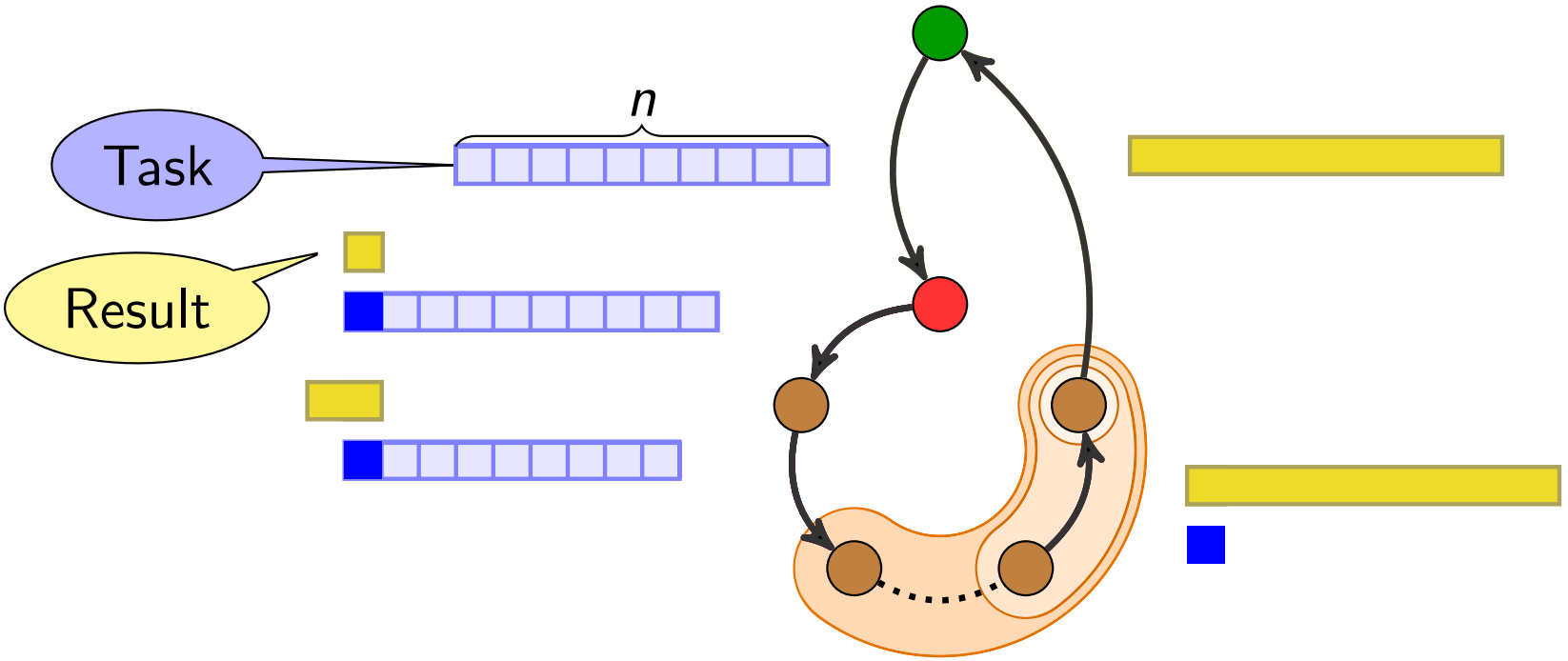
Ring Workers [Arts and Dam, 1999]



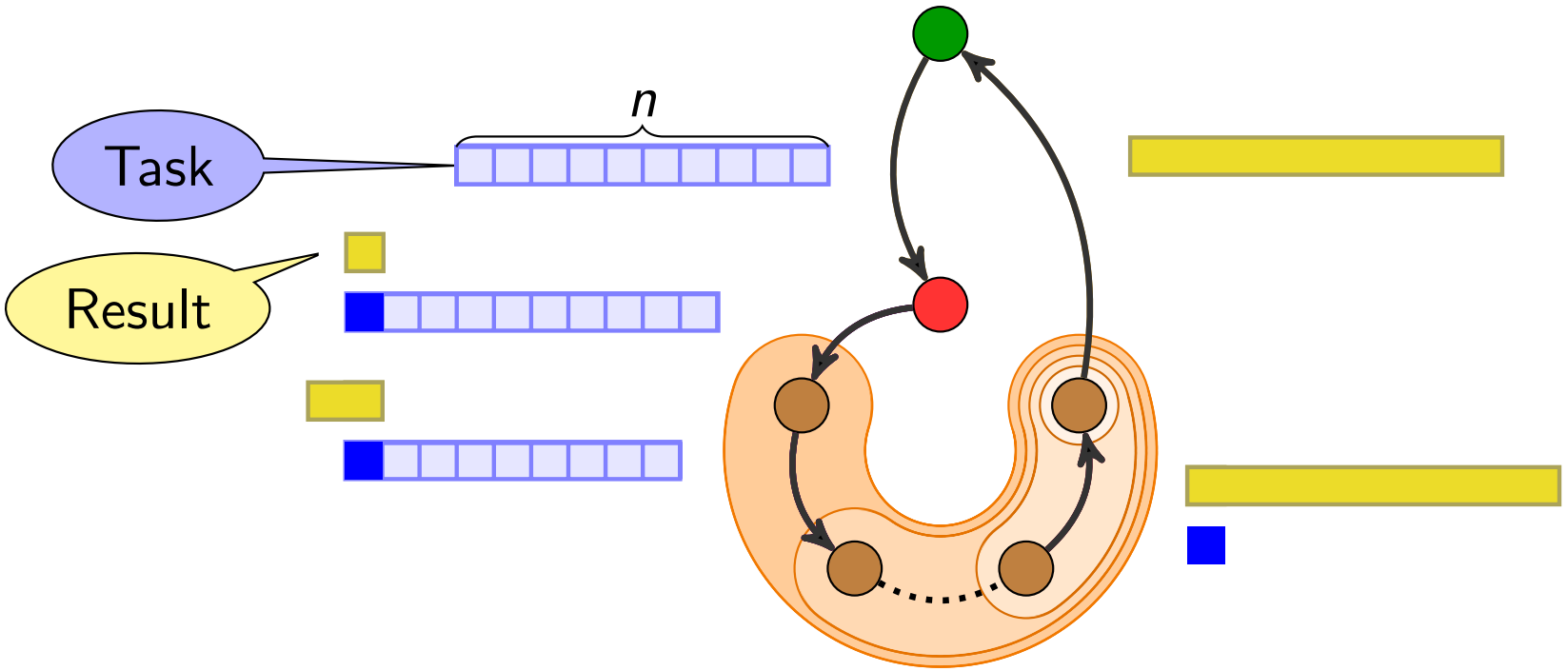
Ring Workers [Arts and Dam, 1999]



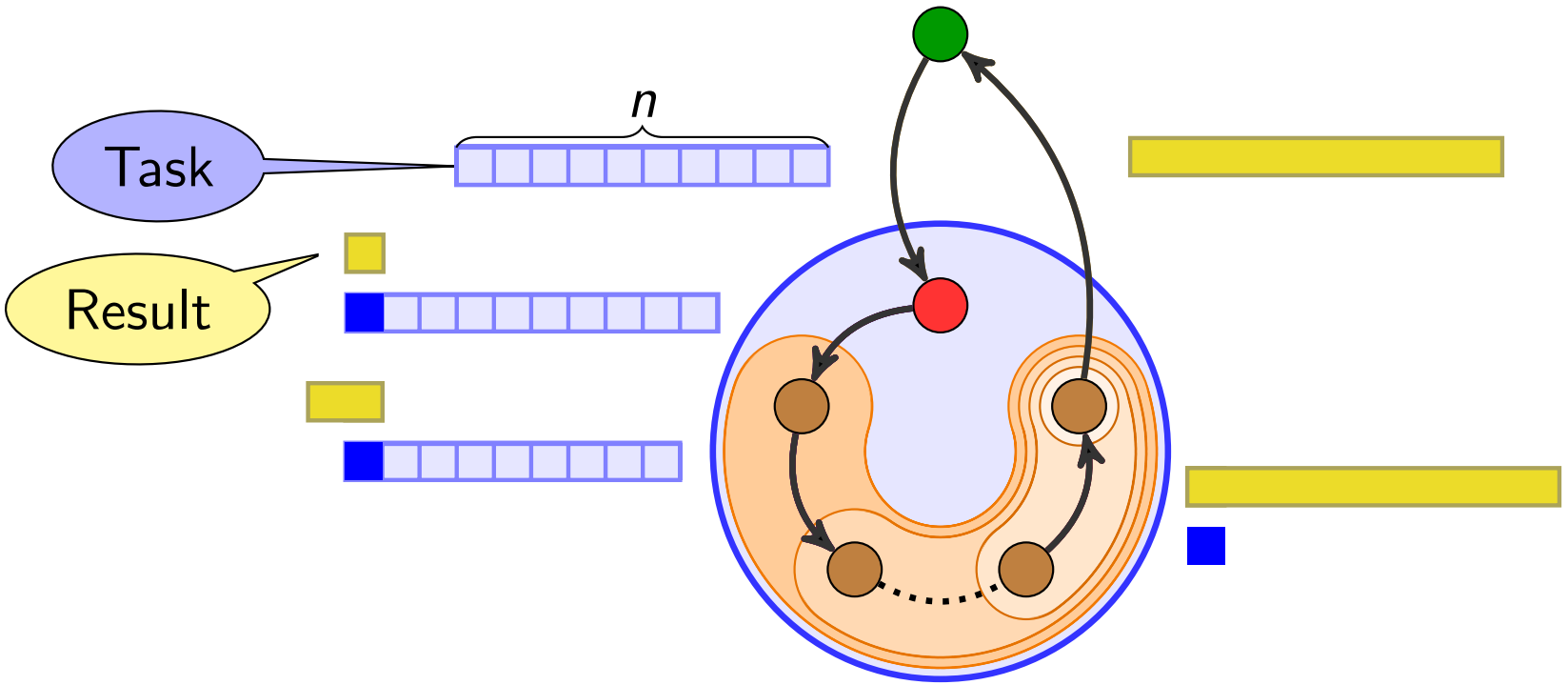
Ring Workers [Arts and Dam, 1999]



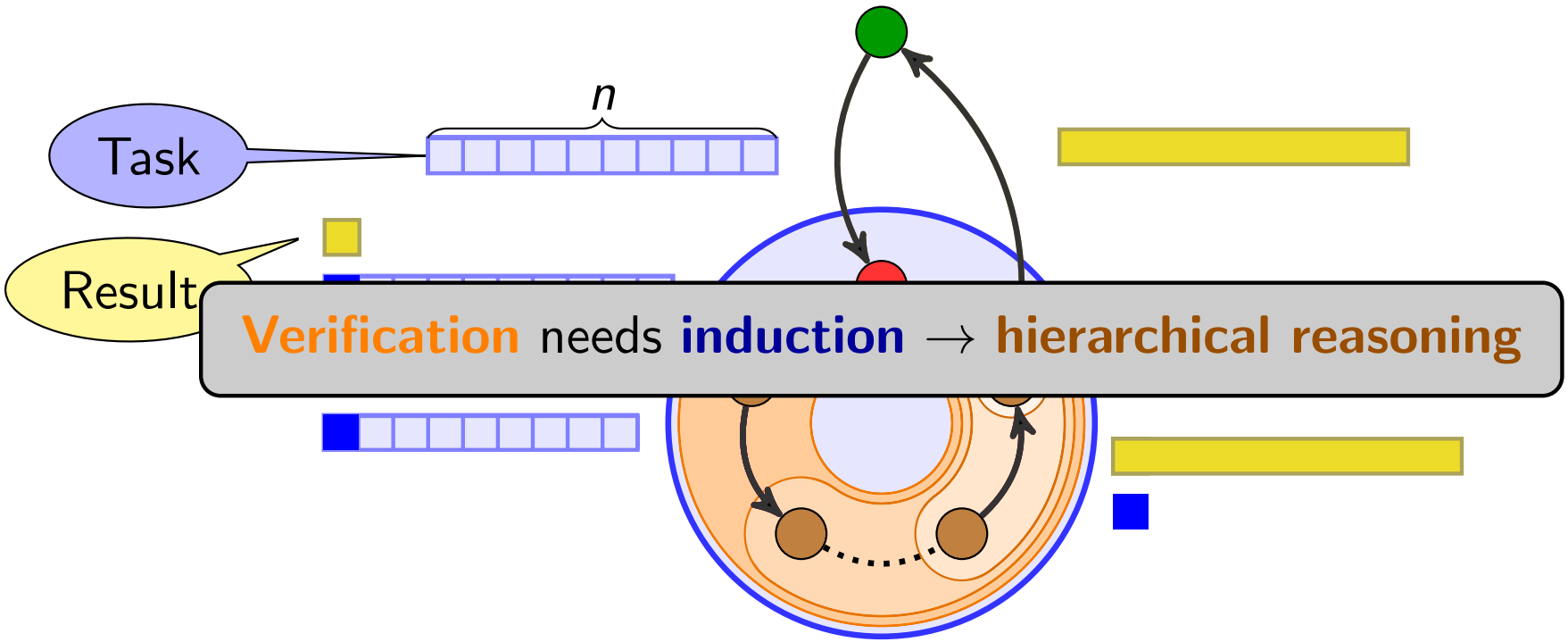
Ring Workers [Arts and Dam, 1999]



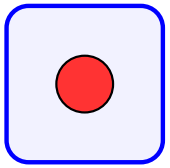
Ring Workers [Arts and Dam, 1999]



Ring Workers [Arts and Dam, 1999]

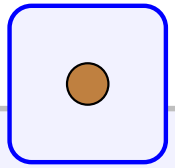


Server Actor



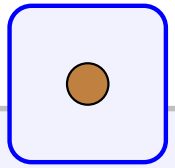
```
actor spec AServer {
   this.req( c, t ) ==> w ← new AWorker;
                        w.do( t, taskSize(t) );
                        w.propagateResult( null, c )
   assumes c ≠ null
}
```

Worker Actor



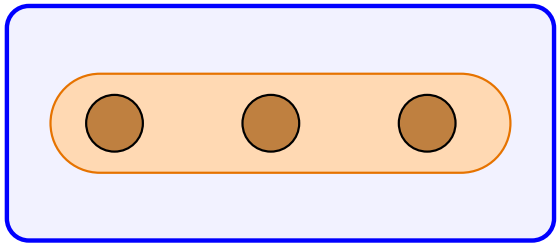
```
actor spec AWorker {
  Value  myResult = null;
  Worker nextWorker = null;

  this.do( t, n ) ==> empty
  assumes n = 1
  asserts myResult = compute(t) ∧ nextWorker = null

  this.do( t, n ) ==> w ← new AWorker;
                      w.do( restTask(t, n), n-1 )
  assumes n > 1
  asserts myResult = compute( firstTask(t, n) ) ∧ nextWorker = w
```

Worker Actor



```
actor spec AWorker {
  Value  myResult = null;
  Worker nextWorker = null;

  this.do( t, n ) ==> empty
  assumes n = 1
  asserts myResult = compute(t) ∧ nextWorker = null

  this.do( t, n ) ==> w ← new AWorker;
                      w.do( restTask(t, n), n-1 )
  assumes n > 1
  asserts myResult = compute( firstTask(t, n) ) ∧ nextWorker = w

  this.propagateResult( v, c ) ==> c.resp( merge(myResult,v) )
  assumes myResult ≠ null && nextWorker = null

  this.propagateResult( v, c ) ==>
                nextWorker.propagateResult( merge(myResult,v), c )
  assumes myResult ≠ null && nextWorker ≠ null
}
```

Worker Group



```
group spec AWorker {
  boolean  working = false;
  CompTask myTask = emptyTask;

  this.do(t, n) ==> empty
  assumes  n ≥ 1
  asserts  myTask = t ∧ working = true

  this.propagateResult(v, c) ==> c.resp(merge(compute(mytask), v))
  assumes  working
}
```

Verification Cases

1. **Composititon** of group and actor specifications
2. **Induction** over parameter values

Verification Cases

1. **Composititon** of group and actor specifications
   - Server actor + Worker group → Server group
2. **Induction** over parameter values
   - Worker actor + Worker group → Worker group

Server Actor + Worker Group → Server Group

```
group spec AServer {
   this.request(c, t) ==> c.response( compute(t) )
   assumes  c != null
}
```

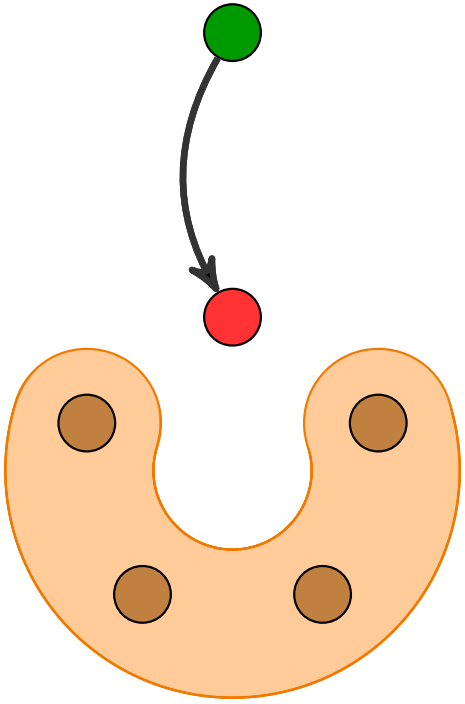
Server Actor + Worker Group → Server Group

```
group spec AServer {
   this.request(c, t) ==> c.response( compute(t) )
   assumes  c != null
}
```

1. this.request(c, t)

Server Actor + Worker Group → Server Group

```
group spec AServer {
   this.request(c, t) ==> c.response( compute(t) )
   assumes  c != null
}
```



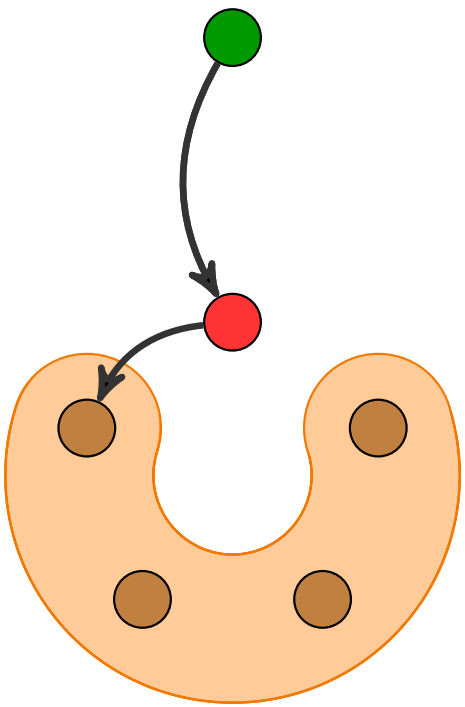
1. this.request(c, t)

Server actor

```
working = false
myTask = emptyTask
```

Server Actor + Worker Group → Server Group

```
group spec AServer {
   this.request(c, t) ==> c.response( compute(t) )
   assumes  c != null
}
```



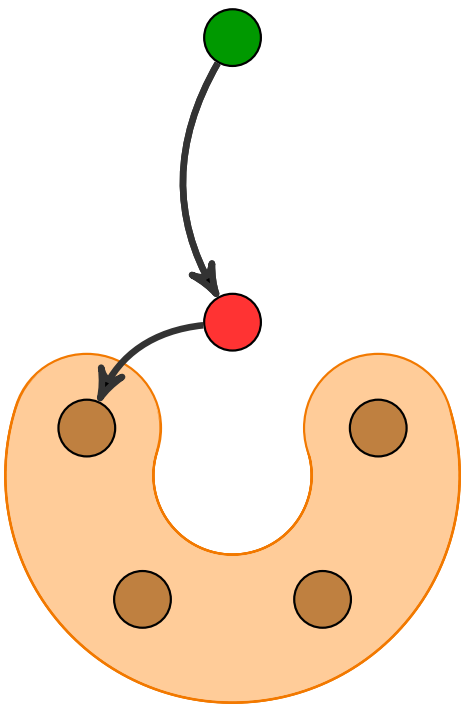
1. this.request(c, t)
2. w.do(t, taskSize(t))

Server actor

```
working = false
myTask = emptyTask
```

Server Actor + Worker Group → Server Group

```
group spec AServer {
   this.request(c, t) ==> c.response( compute(t) )
   assumes  c != null
}
```



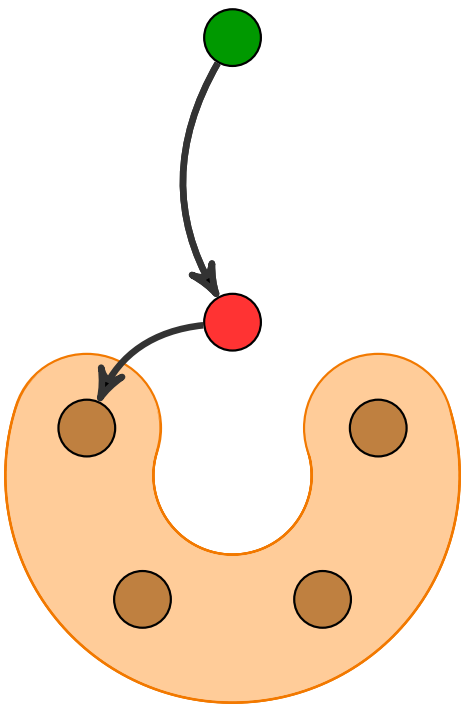
1. this.request(c, t)
2. w.do(t, taskSize(t))

Worker group

```
working = true
myTask = t
```

Server Actor + Worker Group → Server Group

```
group spec AServer {
   this.request(c, t) ==> c.response( compute(t) )
   assumes  c != null
}
```



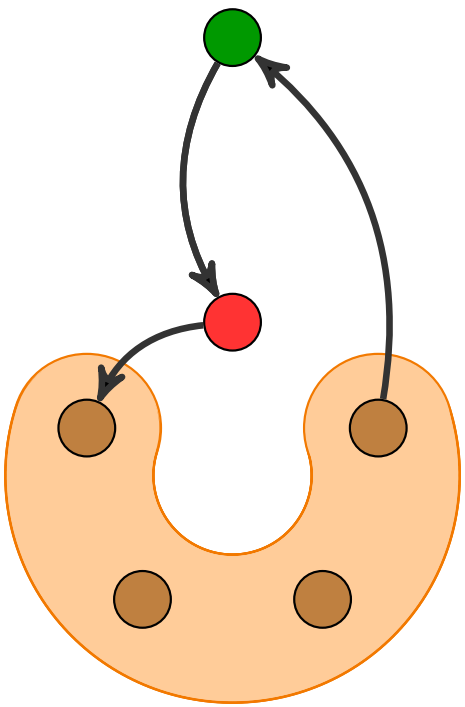
1. this.request(c, t)
2. w.do(t, taskSize(t))
3. w.propagateResult(null, c)

Server actor

```
working = true
myTask = t
```

Server Actor + Worker Group → Server Group

```
group spec AServer {
   this.request(c, t) ==> c.response( compute(t) )
   assumes  c != null
}
```



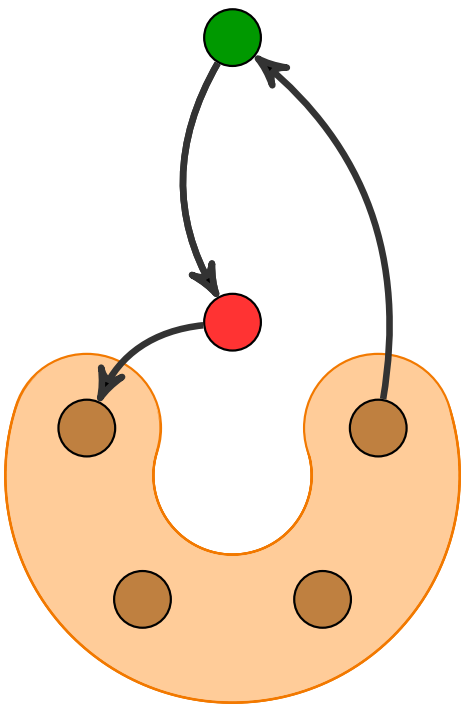
1. this.request(c, t)
2. w.do(t, taskSize(t))
3. w.propagateResult(null, c)
4. c.response(merge(compute(w.myTask), null))

Worker group

```
working = true
myTask = t
```

Server Actor + Worker Group → Server Group

```
group spec AServer {
   this.request(c, t) ==> c.response( compute(t) )
   assumes  c != null
}
```



1. this.request(c, t)
2. w.do(t, taskSize(t))
3. w.propagateResult(null, c)
4. c.response(compute(t))

Worker group

```
working = true
myTask = t
```

Worker Actor + Worker Group → Worker Group

Goals: `do(t, n) ==>` change state

```
propResult(v, c) ==> c.response(merge(compute(mytask), v))
```

Worker Actor + Worker Group → Worker Group

Goals: `do(t, n) ==>` change state

```
propResult(v, c) ==> c.response(merge(compute(mytask), v))
```

Worker Actor + Worker Group → Worker Group

Goals: `do(t, n) ==>` change state

```
propResult(v, c) ==> c.response(merge(compute(mytask), v))
```

Worker Actor + Worker Group → Worker Group

Goals: `do(t, n) ==>` change state

```
propResult(v, c) ==> c.response(merge(compute(mytask), v))
```

Worker Actor + Worker Group → Worker Group

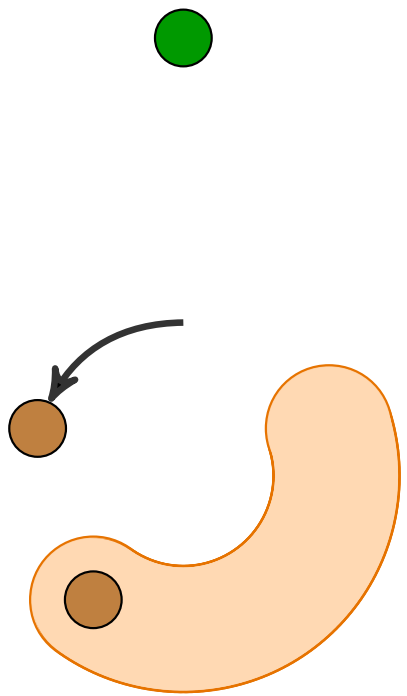
Goals: `do(t, n) ==>` change state

```
propResult(v, c) ==> c.response(merge(compute(mytask), v))
```

Worker Actor + Worker Group → Worker Group

Goals: `do(t, n) ==>` change state

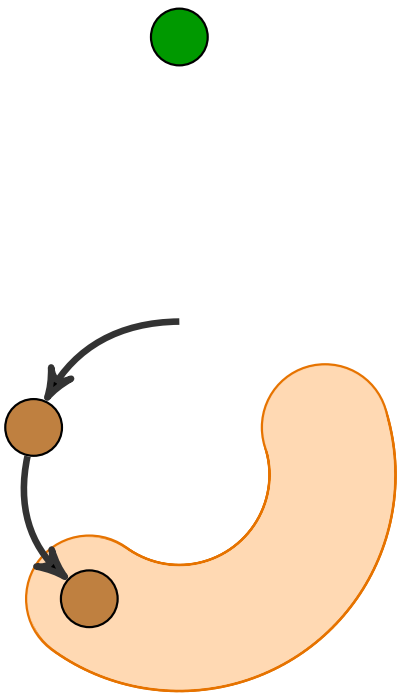
```
propResult(v, c) ==> c.response(merge(compute(mytask), v))
```



Worker Actor + Worker Group → Worker Group

Goals: `do(t, n) ==>` change state

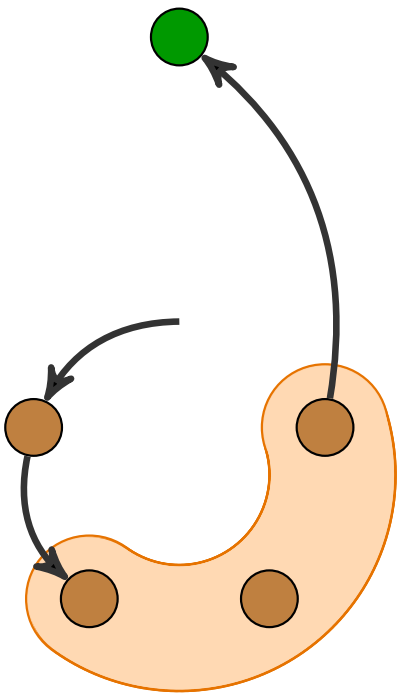
```
propResult(v, c) ==> c.response(merge(compute(mytask), v))
```



Worker Actor + Worker Group → Worker Group

Goals: `do(t, n) ==>` change state

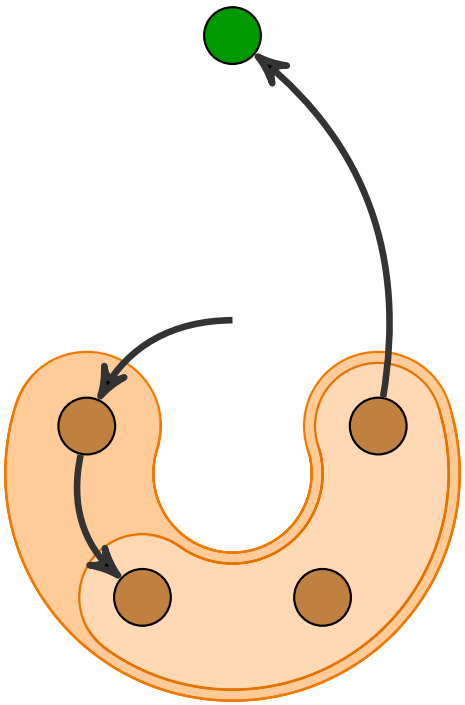
```
propResult(v, c) ==> c.response(merge(compute(mytask), v))
```



Worker Actor + Worker Group → Worker Group

Goals: `do(t, n) ==>` change state

```
propResult(v, c) ==> c.response(merge(compute(mytask), v))
```



Conclusion

- Strong causality + hierarchical reasoning is needed at system level
- Semantic enrichment may lead to easier reasoning

Conclusion

- Strong causality + hierarchical reasoning is needed at system level
- Semantic enrichment may lead to easier reasoning

Future Work

- Multiple entry points for actor groups
- Logic to handle strong causality and hierarchical reasoning