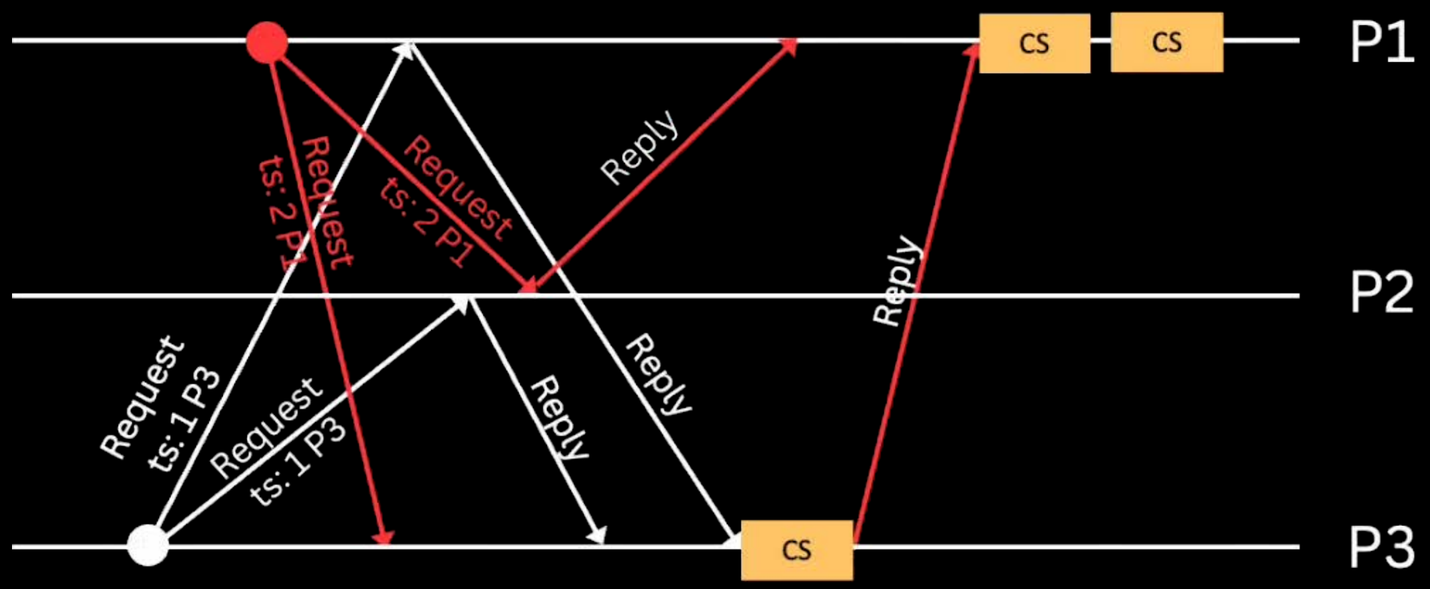
SUZUKI-KASAMI ALGORITHM

A token-based concurrency algorithm for achieving mutual exclusion in distributed systems.

COMP3151 22T3 ASSIGNMENT 2
JINGHAN WANG

Ricart-Agrawala Algorithm



Deficiency

- High communication complexity.
- Relying on global clock synchronization
- Not guarantee fairness

Ichiro Suzuki
Tadao Kasami
Nov 1985

Suzuki-Kasami Algorithm Overcome

Privilege Token

Only the process holding the token can enter the critical section, which ensures mutual exclusion.

Lower Communication Complexity

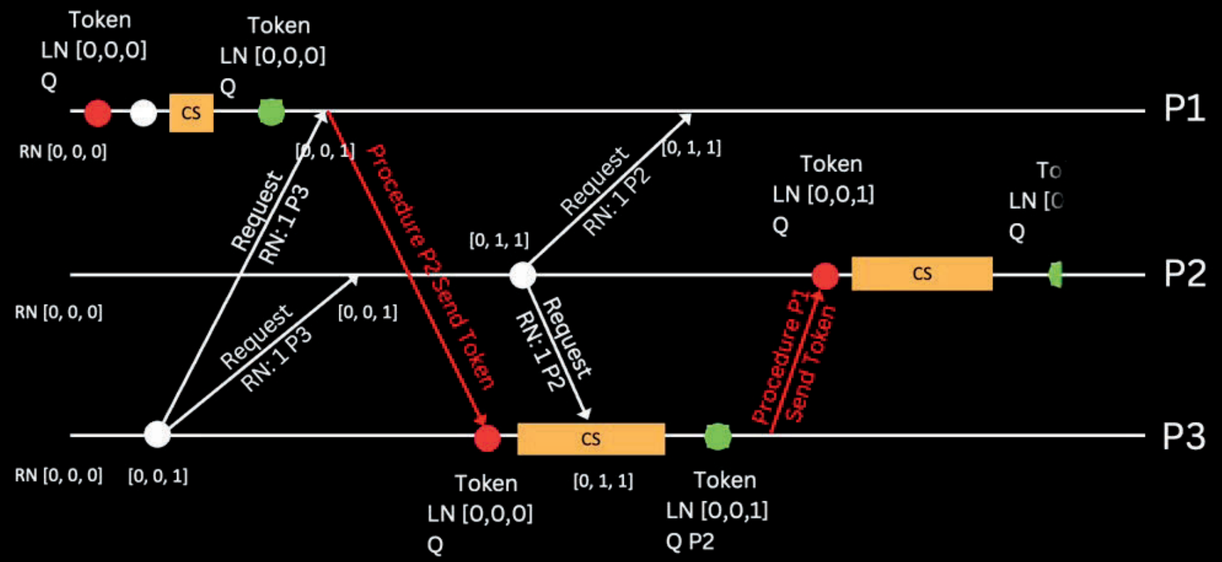
Only N communication per request, significantly reducing overhead.

No Global Clock Synchronization

Not rely on global clock synchronization.

Algorithmic Analysis

```
const I: Integer;   (* the identifier of this node *)
      var HavePrivilege, Requesting:bool;
      j, n: integer;
      Q: queue of integer;
      RN, LN: array[1 .. N] of integer;
      (* The initial values of the variables are:
         HavePrivilege = true in node 1, false in all other nodes;
         Requesting = false;
         Q = empty;
         RN[j] = −1, j = 1, 2, . . . , N;
         LN[j] = −1, j = 1, 2, . . . , N; *)
```




```
procedure P1;
begin
   Requesting := true;
   if not HavePrivilege then
      begin
         RN[I] := RN[I] + 1;
         for all j in {1, 2, . . . , N} − {I} do
            Send REQUEST(I, RN[I]) to node j;
         Wait until PRIVILEGE(Q, LN) is received;
         HavePrivilege := true
      end;
   Critical Section;
   LN[I] := RN[I];
   for all j in {1, 2, . . . , N} − {I} do
      if not in(Q, j) and (RN[j] = LN[j] + 1) then Q := append(Q, j);
   if Q ≠ empty then
      begin
         HavePrivilege := false;
         Send PRIVILEGE(tail(Q), LN) to node head(Q)
      end;
   Requesting := false
end;
```

```
procedure P2;   (* REQUEST(j, n) is received; P2 is indivisible *)
begin
   RN[j] := max(RN[j], n);
   if HavePrivilege and not Requesting and (RN[j] = LN[j] + 1) then
      begin
         HavePrivilege := false;
         Send PRIVILEGE(Q, LN) to node j
      end
end;
```

Go Language

PUBLICLY IN 2009 BY GOOGLE

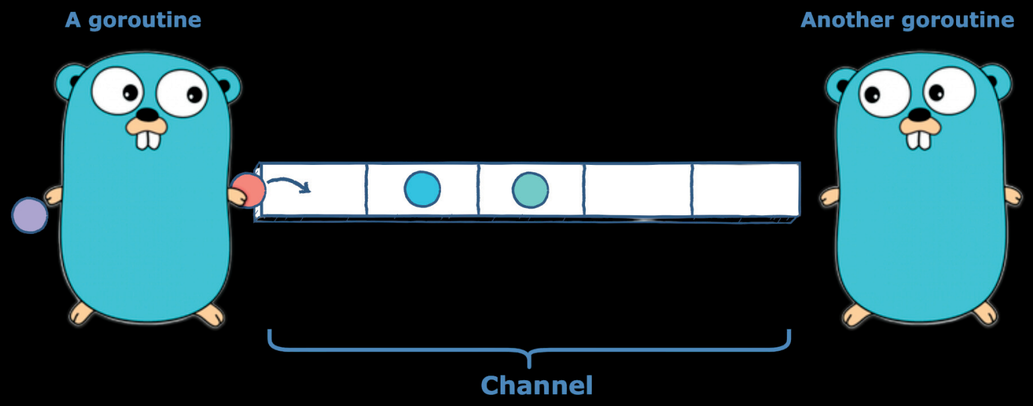
A statically typed, compiled high-level programming language

```
fmt.Print("Hello, Go")
```

Goroutine

LIGHT-WEIGHT PROCESS

- Provide the goroutine mechanism as a native concurrency mechanism
- Each goroutine requires very little memory



Channel

INFORMATION EXCHANGE BETWEEN GOROUTINES

- Synchronous channel
- Asynchronous channel

Data Structure

```
type process struct {
    id          int
    request     bool
    token       *token
    RN          []int
    requestChan []chan request
    tokenChan   []chan *token
    mu          sync.Mutex
}
```

```
const I: Integer;   (* the identifier of this node *)
      var HavePrivilege, Requesting:bool;
      j, n: integer;
      Q: queue of integer;
      RN, LN: array[1 .. N] of integer;
      (* The initial values of the variables are:
         HavePrivilege = true in node 1, false in all other nodes;
         Requesting = false;
         Q = empty;
         RN[j] = −1, j = 1, 2, . . . , N;
         LN[j] = −1, j = 1, 2, . . . , N; *)
```

```
type request struct {
    sendId  int
    content int
}
```

```
type token struct {
    LN []int
    Q  queue
}
```

```
type queue struct {
    items []int
}

func (q *queue) Enqueue(i int) {
    q.items = append(q.items, i)
}

func (q *queue) Dequeue() int {
    toRemove := q.items[0]
    q.items = q.items[1:]
    return toRemove
}

func (q *queue) find(i int) bool {
    for _, v := range q.items {
        if v == i : true ⤴
    }
    return false
}

func (q *queue) isEmpty() bool {
    return len(q.items) == 0
}
```

Data Structure

```go
type process struct {
    id          int
    request     bool
    token       *token
    RN          []int
    requestChan []chan request
    tokenChan   []chan *token
    mu          sync.Mutex
}
```

```
const l: Integer;   (* the identifier of this node *)
      var HavePrivilege, Requesting:bool;
      j, n: integer;
      Q: queue of integer;
      RN, LN: array[1 .. N] of integer;
      (* The initial values of the variables are:
         HavePrivilege = true in node 1, false in all other nodes;
         Requesting = false;
         Q = empty;
         RN[j] = −1, j = 1, 2, . . . , N;
         LN[j] = −1, j = 1, 2, . . . , N; *)
```

```go
type request struct {
    sendId  int
    content int
}
```

```go
type token struct {
    LN []int
    Q  queue
}
```

```go
type queue struct {
    items []int
}

func (q *queue) Enqueue(i int) {
    q.items = append(q.items, i)
}

func (q *queue) Dequeue() int {
    toRemove := q.items[0]
    q.items = q.items[1:]
    return toRemove
}

func (q *queue) find(i int) bool {
    for _, v := range q.items {
        if v == i : true ⤴
    }
    return false
}

func (q *queue) isEmpty() bool {
    return len(q.items) == 0
}
```

Initialization

```go
func main() {
    const number = 5
    systemToken := new(token)
    systemToken.LN = make([]int, number)
    systemToken.Q = queue{}

    processes := make([]*process, number)
    requestChan := make([]chan request, number)
    tokenChan := make([]chan *token, number)

    for i := 0; i < number; i++ {
        // Accept Request Channel: As not block no matter how many
        // requests the port accepts, set asynchronous channel and
        //give them ample cache space.
        requestChan[i] = make(chan request, 1000)
        // The algorithm set the token is not loss, set it as a
        // synchronous channel to ensure certain acquisitions.
        tokenChan[i] = make(chan *token)
        systemToken.LN[i] = -1
    }
```

```go
    for i := 0; i < number; i++ {
        processes[i] = &process{
            id:          i,
            request:     false,
            token:       nil,
            RN:          make([]int, number),
            requestChan: requestChan,
            tokenChan:   tokenChan,
        }
        for j := range processes[i].RN {
            processes[i].RN[j] = -1
        }
    }

    // Randomly set the token at any node
    processes[rand.Intn(number)].token = systemToken

    for i := 0; i < number; i++ {
        go processes[i].Receive()
        go processes[i].Send()
    }

    select {}
}
```

Initialization

```go
func main() {
    const number = 5
    systemToken := new(token)
    systemToken.LN = make([]int, number)
    systemToken.Q = queue{}

    processes := make([]*process, number)
    requestChan := make([]chan request, number)
    tokenChan := make([]chan *token, number)

    for i := 0; i < number; i++ {
        // Accept Request Channel: As not block no matter how many
        // requests the port accepts, set asynchronous channel and
        //give them ample cache space.
        requestChan[i] = make(chan request, 1000)
        // The algorithm set the token is not loss, set it as a
        // synchronous channel to ensure certain acquisitions.
        tokenChan[i] = make(chan *token)
        systemToken.LN[i] = -1
    }
```

```go
    for i := 0; i < number; i++ {
        processes[i] = &process{
            id:          i,
            request:     false,
            token:       nil,
            RN:          make([]int, number),
            requestChan: requestChan,
            tokenChan:   tokenChan,
        }
        for j := range processes[i].RN {
            processes[i].RN[j] = -1
        }
    }

    // Randomly set the token at any node
    processes[rand.Intn(number)].token = systemToken

    for i := 0; i < number; i++ {
        go processes[i].Receive()
        go processes[i].Send()
    }

    select {}
}
```

Send
(Enter critical section)

```go
func (p *process) Send() {
    for {
        // Randomly sleep for a while to simulate the process delay
        time.Sleep(time.Duration(rand.Intn( n: 100)) * time.Millisecond)
        p.mu.Lock()
        p.request = true
        if p.token == nil {
            p.RN[p.id]++
            for i := range p.requestChan {
                if i != p.id {
                    // Simulation of network uncertainty in a distributed scenario,
                    // 1/50 probability of message loss
                    if rand.Intn( n: 50) != 0 {
                        // Randomly sleep for a while to simulate the network delay
                        time.Sleep(time.Duration(rand.Intn( n: 100)) * time.Millisecond)
                        p.requestChan[i] <- request{ sendId: p.id,  content: p.RN[p.id]}
                    } else {
                        fmt.Println( a...: "Process", p.id, "Requests Message loss")
                    }
                }
            }
            p.token = <-p.tokenChan[p.id]
            fmt.Println( a...: "Process", p.id, "model1Receive token")
        }
        p.mu.Unlock()
        fmt.Println( a...: "Process", p.id, "enter CS")

        p.mu.Lock()
        p.token.LN[p.id] = p.RN[p.id]
        for i := range p.token.LN {
            if i != p.id && !p.token.Q.find(i) && p.RN[i] == p.token.LN[i]+1 {
                p.token.Q.Enqueue(i)
            }
        }

        if !p.token.Q.isEmpty() {
            sendId := p.token.Q.Dequeue()
            p.tokenChan[sendId] <- p.token
            p.token = nil
            fmt.Println( a...: "Send Process", p.id, "model1_send token to", sendId)
        }

        p.request = false
        p.mu.Unlock()
    }
}
```

Receive (Receive Requests)

After the accepting process has requested the lock, set to process all requests in the channel to completion

```go
func (p *process) Receive() {
    receiveChan := p.requestChan[p.id]
    for {
        p.mu.Lock()
        for len(receiveChan) > 0 {
            m := <-receiveChan

            if p.RN[m.sendId] < m.content {
                p.RN[m.sendId] = m.content
            }

            if p.token != nil && !p.request && p.RN[m.sendId] == p.token.LN[m.sendId]+1 {
                p.tokenChan[m.sendId] <- p.token
                p.token = nil
                fmt.Println( a...: "Receive Process", p.id, "model1_send token to", m.sendId)
            }
        }
        p.mu.Unlock()
    }
}
```

Suzuki
Kasami
Algorithm

Concurrency Properties

Safety Properties

- Only the process of possessing the token can enter the critical section at anytime.

Liveness Property

- No deadlock: Any process that requests the critical section will eventually receive the token, as the token is always passed to the following process in the request queue.
- No starvation: The use of a queue ensures fairness, avoiding starvation.

Suzuki
Kasami
Algorithm

A theoretical model for study

Disadvantage or Limitation

Reliance on Reliable Communication

The system is at risk of failure due to its dependence on reliable communication. The loss of the token could incapacitate the entire system.

Awareness of All Processes

The requirement for each process to be aware of all other processes can be impractical, especially in large-scale systems.

Token Allocation

While the Suzuki-Kasami algorithm uses a queue to store requests, it does not specify how to select the next process to receive the token. This leaves the fairness issue found in the Ricart-Agrawala algorithm unaddressed.


THANKS FOR
WATCHING