Tema 6. Sistemas Distribuidos

Jorge García Duque
http://www.det.uvigo.es/~jgd
Depto. Enxeñería Telemática
Universidad de Vigo

Introducción (I)

- Sistema Distribuido: conjunto de nodos (ordenadores) conectados en red cuyos componentes (hardware o software) se comunican y coordinan únicamente mediante el intercambio de mensajes:
  - Concurrencia.
  - Ausencia de reloj global.
  - Fallos independientes (red y/o de un nodo).
- Motivación: compartir recursos (impresoras, discos, ficheros, bases de datos, audio, vídeo, servicios, etc.).

Introducción (II)

- Retos en el diseño de sistemas distribuidos:
  - Heterogeneidad.
  - Extensibilidad.
  - Seguridad.
  - Escalabilidad.
  - Concurrencia.
  - Tratamiento de Fallos.
    - Detección.
    - Tolerancia.
    - Recuperación (*rollback).*
    - Redundancia.
  - Transparencia.
    - Acceso.
    - Localización.
    - Concurrencia.
    - Replicación.
    - Movilidad.
    - Prestaciones.
    - Escalado.

Exclusión Mutua Distribuida

- Suposiciones:
  - Los nodos siempre pueden recibir y enviar mensajes.
  - Cada nodo dispone de un canal de comunicaciones con cada uno de los otros nodos.
  - Los mensajes se intercambian sin error (aunque pueden entregarse en un orden diferente).
  - Los mensajes siempre se entregan, aunque el tiempo puede variar arbitrariamente.
  - Primitivas *send* y *receive* bloqueantes:

```
send(tipo, id_destino, datos);  receive(tipo, datos);
```

¿cómo se sabe quién ha enviado el mensaje?

- Algoritmos:
  - Centralizado (nodo coordinador).
  - Distribuidos:
    - *Ricart-Agrawala*: basado en solicitud.
    - Paso de Testigo: *Ricart-Agrawala, Neilsen-Mizuno.*

Algoritmo *Ricart-Agrawala* (I)

- Los nodos intercambian *tickets* de petición.
- El *ticket* menor entra en la sección crítica.
- Cada nodo debe conocer la *dirección* (*id_destino)* del resto de los nodos.
- Nodos estáticos: cada nodo debe conocer el resto de nodos que compiten por la exclusión mutua.
- Dos tipos de mensajes: *request* y *reply.*

¿puede haber varios procesos en un mismo nodo?

Algoritmo *Ricart-Agrawala* (II)

```
int mi_ticket = 0, mi_id = 1234, id_nodos_pend[N-1] = {0}, num_pend = 0;
```

```
Process Main
  int id_aux = 0, i = 0, id_nodos[N-1] = {1235, 1236, …};
  while (1) {
p1: ……;
p2: mi_ticket = random();
p3: for (i = 0; i < N-1; i++) send(REQUEST, id_nodos[i], {mi_id, mi_ticket});
p4: for (i = 0; i < N-1; i++) receive(REPLY, &id_aux);
p5: SECCIÓN CRÍTICA;
p6: for (i = 0; i < num_pend; i++)
      send(REPLY, id_nodos_pend[i], mi_id);
p7: num_pend = 0;
  }

Process Receptor
  int id_nodo_origen = 0, ticket_origen = 0;
  while (1) {
q1: receive(REQUEST, {&id_nodo_origen, &ticket_origen);
q2: if (ticket_origen < mi_ticket) send(REPLY, id_nodo_origen, mi_id);
q3: else id_nodos_pend[num_pend++] = id_nodo_origen;
  }
```

Algoritmo *Ricart-Agrawala* (III)

```
int mi_ticket = 0, mi_id = 1234, id_nodos_pend[N-1] = {0}, num_pend = 0;
```

```
Process Main
  int id_aux = 0, i = 0, id_nodos[N-1] = {1235, 1236, …};
  while (1) {
p1: ……;
p2: mi_ticket = random();
p3: for (i = 0; i < N-1; i++) send(REQUEST, id_nodos[i], {mi_id, mi_ticket});
p4: for (i = 0; i < N-1; i++) receive(REPLY, &id_aux);
p5: SECCIÓN CRÍTICA;
p6: for (i = 0; i < num_pend; i++)
      send(REPLY, id_nodos_pend[i], mi_id);
p7: num_pend = 0;
  }
```

Mismo Ticket

```
or
q2: if (ticket_origen < mi_ticket OR
        (ticket_origen == mi_ticket AND (id_nodo_origen < mi_id))
q2: if (ticket_origen < mi_ticket) send(REPLY, id_nodo_origen, mi_id);
q3: else id_nodos_pend[num_pend++] = id_nodo_origen;
  }
```

Algoritmo *Ricart-Agrawala* (IV)

```
int mi_ticket = 0, mi_id = 1234, id_nodos_pend[N-1] = {0}, num_pend = 0;
```

Ticket arbitrario

```
= 0, i = 0, id_nodos[N-1] = {1235, 1236, …};
while (1) {
p2: mi_ticket = max_ticket + 1;
p2: mi_ticket = random();
p3: for (i = 0; i < N-1; i++) send(REQUEST, id_nodos[i], {mi_id, mi_ticket});
p4: for (i = 0; i < N-1; i++) receive(REPLY, &id_aux);
p5: SECCIÓN CRÍTICA;
p6: for (i = 0; i < num_pend; i++)
      send(REPLY, id_nodos_pend[i], mi_id);
p7: num_pend = 0;
  }

Process Receptor
  int id_nodo_origen = 0, ticket_origen = 0;
  while (1) {
q1: receive(REQUEST, {&id_nodo_origen, &ticket_origen);
q1.5: max_ticket = MAX(max_ticket, ticket_origen);
q3: else id_nodos_pend[num_pend++] = id_nodo_origen;
  }
```

Algoritmo *Ricart-Agrawala* (V)

```
int mi_ticket = 0, mi_id = 1234, id_nodos_pend[N-1] = {0}, num_pend = 0;

Process Main
  int id_aux = 0, i = 0, id_nodos[N-1] = {1235, 1236, …};
  ...
p1.5: quiero = 1;
p3: for (i = 0; i < N-1; i++) send(REQUEST, id_nodos[i], {mi_id, mi_ticket});
p4: for (i = 0; i < N-1; i++) receive(REPLY, &id_aux);
p5: SECCIÓN CRÍTICA;
p5.5: quiero = 0;
      send(REPLY, id_nodos_pend[i], mi_id);
p7: num_pend = 0;
  }

Process Receptor
q2: if (NOT quiero OR ticket_origen < mi_ticket OR
        (ticket_origen == mi_ticket AND (id_nodo_origen < mi_id))
q2: if (ticket_origen < mi_ticket) send(REPLY, id_nodo_origen, mi_id);
q3: else id_nodos_pend[num_pend++] = id_nodo_origen;
  }
```

Nodo perezoso

Algoritmo *Ricart-Agrawala* (VI)

```
int mi_ticket = 0, mi_id = 1234, id_nodos_pend[N-1] = {0}, num_pend = 0, quiero = 0, max_ticket = 0;
```

```
Process Main
  int id_aux = 0, i = 0, id_nodos[N-1] = {1235, 1236, …};
  while (1) {
p1: ……;
p2: quiero = 1;
p3: mi_ticket = max_ticket + 1;
p4: for (i = 0; i < N-1; i++) send(REQUEST, id_nodos[i], {mi_id, mi_ticket});
p5: for (i = 0; i < N-1; i++) receive(REPLY, &id_aux);
p6: SECCIÓN CRÍTICA;
p7: quiero = 0;
p8: for (i = 0; i < num_pend; i++)
      send(REPLY, id_nodos_pend[i], mi_id);
p9: num_pend = 0;
  }

Process Receptor
  int id_nodo_origen = 0, ticket_origen = 0;
  while (1) {
q1: receive(REQUEST, {&id_nodo_origen, &ticket_origen);
q2: max_ticket = MAX(max_ticket, ticket_origen);
q3: if (NOT quiero OR ticket_origen < mi_ticket OR
        (ticket_origen == mi_ticket AND (id_nodo_origen < mi_id))
        send(REPLY, id_nodo_origen, mi_id);
q4: else id_nodos_pend[num_pend++] = id_nodo_origen;
  }
```

Algoritmo *Ricart-Agrawala* (VII)

```
int mi_ticket = 0, mi_id = 1234, id_nodos_pend[N-1] = {0}, num_pend = 0, quiero = 0, max_ticket = 0;

Process Main
  int id_aux = 0, i = 0, id_nodos[N-1] = {1235, 1236, …};
  while (1) {
p1: ……;
p2: quiero = 1;
p3: mi_ticket = max_ticket + 1;
p4: for (i = 0; i < N-1; i++) send(REQUEST, id_nodos[i], {mi_id, mi_ticket});
p5: for (i = 0; i < N-1; i++) receive(REPLY, &id_aux);
p6: SECCIÓN CRÍTICA;
p7: quiero = 0;
p8: for (i = 0; i < num_pend; i++)
      send(REPLY, id_nodos_pend[i], mi_id);
p9: num_pend = 0;
  }

Process Receptor
  int id_nodo_origen = 0, ticket_origen = 0;
  while (1) {
q1: receive(REQUEST, {&id_nodo_origen, &ticket_origen);
q2: max_ticket = MAX(max_ticket, ticket_origen);
q3: if (NOT quiero OR ticket_origen < mi_ticket OR
        (ticket_origen == mi_ticket AND (id_nodo_origen < mi_id))
        send(REPLY, id_nodo_origen, mi_id);
q4: else id_nodos_pend[num_pend++] = id_nodo_origen;
  }
```

Mutex

Mutex

Algoritmo de Paso de Testigo (I)

- Existe un único testigo que se pasa entre los procesos (inicialmente a un proceso cualquiera).
- El que tiene el testigo puede entrar en la sección crítica (reteniendo el testigo). El resto deben esperar.
- Es necesario establecer un anillo lógico entre los procesos.

¿a quién se pasa el testigo?

¿qué sucede si se pierde el testigo?

¿y si falla un proceso?

Algoritmo de Paso de Testigo (II)

- *Ricart-Agrawala:*
  - No existe un anillo lógico predefinido, sino que se establece en base a los procesos que demandan entrar en la sección crítica.
  - Cada proceso envía un mensaje de petición (*request)* a todos los procesos con un identificador de la petición (igual que en el algoritmo anterior).
  - Con el testigo se envía un vector con el identificador de la última petición atendida de cada proceso.
  - Cuando un proceso recibe el testigo y entra en la sección crítica actualiza el vector de peticiones atendidas en la posición correspondiente a dicho proceso.
  - Un proceso puede retener el testigo aun no estando en la sección crítica si después de acceder a su sección crítica no existen peticiones sin atender.
- *Neilsen-Mizuno:*
  - No existe un anillo lógico predefinido, sino que se establece en base a los procesos que demandan entrar en la sección crítica.
  - Las peticiones no se envían a todos los procesos sino a través de rutas virtuales que se van construyendo para dirigir el testigo hacia las peticiones.
  - Algoritmo más eficiente.

Algoritmo de Paso de Testigo (III). *Ricart-Agrawala*

```
int vector_peticiones[N] = {0}, vector_atendidas[N]= {0}, dentro = 0, testigo = 1;// ->0 los otros nodos
```

```
Process Main
  int mi_id = 1234, mi_peticion = 0, i = 0, id_nodo_sig = 0, id_nodos[N-1] = {1235, 1236, …};
  while (1) {
p1: ……;
p2: if (NOT testigo) {
p3:   mi_peticion = mi_peticion + 1;
p4:   for (i = 0; i < N-1; i++) send(REQUEST, id_nodos[i], {mi_id, mi_peticion});
p5:   receive(TOKEN, &vector_antedidas);
p6:   testigo = 1;
   }
p7: dentro = 1;
p8: SECCIÓN CRÍTICA;
p9: vector_atendidas[mi_id] = mi_peticion;
p10: dentro = 0;
p11: if (EXISTS id_nodo_sig / vector_peticiones[id_nodo_sig] > vector_atendidas[id_nodo_sig])
      send_token(id_nodo_sig);
  }
Process Receptor
  int id_nodo_origen = 0, num_peticion_origen = 0;
  while (1) {
q1: receive(REQUEST, {&id_nodo_origen, &num_peticion_origen});
q2: vector_peticiones[id_nodo_origen] = MAX(vector_peticiones[id_nodo_origen], num_peticion_origen);
q3: if (testigo AND (NOT dentro) AND vector_peticiones[id_nodo_origen] > vector_atendidas[id_nodo_origen])
      send_token(id_nodo_origen);
  }
```

Algoritmo de Paso de Testigo (IV). *Ricart-Agrawala*

```
int vector_peticiones[N] = {0}, vector_atendidas[N]= {0}, dentro = 0, testigo = 1;// ->0 los otros nodos
```

```
Process Main
  int mi_id = 1234, mi_peticion = 0, i = 0, id_nodo_sig = 0, id_nodos[N-1] = {1235, 1236, …};
  while (1) {
p1: ……;
p2: if (NOT testigo) {
p3:   mi_peticion = mi_peticion + 1;
p4:   for (i = 0; i < N-1; i++) send(REQUEST, id_nodos[i], {mi_id, mi_peticion});
p5:   receive(TOKEN, &vector_antedidas);
p6:   testigo = 1;
    }
p7: dentro = 1;
p8: SECCIÓN CRÍTICA;
p9: vector_atendidas[mi_id] = mi_peticion;
p10: dentro = 0;
p11: if (EXISTS id_nodo_sig / vector_peticiones[id_nodo_sig] > vector_atendidas[id_nodo_sig])
      send_token(id_nodo_sig);
  }
Process Receptor
  int id_nodo_origen = 0, num_peticion_origen = 0;
  while (1) {
q1: receive(REQUEST, {&id_nodo_origen, &num_peticion_origen});
q2: vector_peticiones[id_nodo_origen] = MAX(vector_peticiones[id_nodo_origen], num_peticion_origen);
q3: if (testigo AND (NOT dentro) AND vector_peticiones[id_nodo_origen] > vector_atendidas[id_nodo_origen])
      send_token(id_nodo_origen);
  }
```

```
send_token(int id_nodo) {
  testigo = 0;
  send(TOKEN, id_nodo, vector_atendidas);
}
```

Algoritmo de Paso de Testigo (V).
Ricart-Agrawala

```
int vector_peticiones[N] = {0}, vector_atendidas[N]= {0}, dentro = 0, testigo = 1;// ->0 los otros nodos
-------------------------------------------------------------------------------------------------------
 Process Main
   int mi_id = 1234, mi_peticion = 0, i = 0, id_nodo_sig = 0, id_nodos[N-1] = {1235, 1236, …};
   while (1) {
 p1: ……;
 p2: if (NOT testigo) {
 p3:    mi_peticion = mi_peticion + 1;
 p4:    for (i = 0; i < N-1; i++) send(REQUEST, id_nodos[i], {mi_id, mi_peticion});
 p5:    receive(TOKEN, &vector_antedidas);
 p6:    testigo = 1;
    }
 p7: dentro = 1;
 p8: SECCIÓN CRÍTICA;
 p9: vector_atendidas[mi_id] = mi_peticion;
p10: dentro = 0;
p11: if (EXISTS id_nodo_sig / vector_peticiones[id_nodo_sig] > vector_atendidas[id_nodo_sig])
       send_token(id_nodo_sig);
   }
Process Receptor
   int id_nodo_origen = 0, num_peticion_origen = 0;
   while (1) {
 q1: receive(REQUEST, {&id_nodo_origen, &num_peticion_origen});
 q2: vector_peticiones[id_nodo_origen] = MAX(vector_peticiones[id_nodo_origen], num_peticion_origen);
 q3: if (testigo AND (NOT dentro) AND vector_peticiones[id_nodo_origen] > vector_atendidas[id_nodo_origen])
       send_token(id_nodo_origen);
   }
```

Mutex

Mutex

Mutex

Mutex

Consenso en Sistemas Distribuidos

- Cada nodo propone un valor y debe decidir en función de los valores recibidos por el resto de nodos.
- Si no hay fallos basta con un algoritmo de votación por mayoría, ya que todos decidirán por el mismo valor y se alcanzará el consenso.
- Se contemplarán dos tipos de fallos:
  - Fallos *Crash*: un nodo deja de enviar mensajes en cualquier momento (se asumen que existe algún mecanismo para detectar la caída de un nodo).
  - Fallos *Bizantinos:* un nodo envía mensajes arbitrarios.

Algoritmo de los Generales Bizantinos (I)

- *"Un grupo de ejércitos Bizantinos rodea una ciudad enemiga. Si se coordinan para atacar juntos vencen, en caso contrario deben retirarse para evitar ser derrotados. Los generales de cada ejército disponen de mensajeros fiables para enviar mensajes de un general a otro. Sin embargo, algunos de los generales pueden ser traidores y no comunicar sus intenciones o incluso enviar mensajes diferentes a cada general (a unos les dice que apoya el ataque y a otros que opta por retirarse)."*
- El objetivo es diseñar un algoritmo para que todos los generales *leales* lleguen a un consenso (atacar o retirarse).
- La decisión se deberá tomar en base a la mayoría, y en caso de empate se optará por la retirada.

Algoritmo de los Generales Bizantinos (II)

- Algoritmo de una ronda:
  1. Se envía un mensaje al resto de nodos.
  2. Se espera respuesta del resto.
  3. Se decide por mayoría.

```
int planes[N-1], mi_plan, plan_final, mi_id, general[N-1], i, id_aux;
```

```
p1: mi_plan = random('A', 'R');
p2: for (i = 0; i < N-1; i++) send(general[i], {mi_id, mi_plan});
p3: for (i = 0; i < N-1; i++) receive(&id_aux, &planes[i]);
p4: plan_final = MAYORÍA(planes, mi_plan);
```

¿es capaz de alcanzarse el consenso en presencia de fallos crash?

Algoritmo de los Generales Bizantinos (III)

- Solución:
  - Es necesario poder distinguir *leales* de *traidores.*
  - Segunda ronda donde cada *general* comunica al resto lo que él recibió de los otros.
  - Votación:
    - Para cada *general,* por mayoría de lo que recibí de él y de lo que me dijeron el resto que recibieron de dicho *general.*
    - Mayoría de los resultados obtenidos en el paso anterior para cada *general* (incluyendo mi voto),

Algoritmo de los Generales Bizantinos (IV)

```
int planes[N-1], planes_otros[N-1][N-2], planes_finales[N-1], mi_plan,
    plan_final, mi_id, general[N-1], i, j, id_aux;
```

```
p1: mi_plan = random('A', 'R');
p2: for (i = 0; i < N-1; i++) send(general[i], {mi_id, mi_plan});
p3: for (i = 0; i < N-1; i++) receive(&id_aux, &planes[i]);
p4: for (i = 0; i < N-1; i++)
     for (j = 0; j < N-1 AND j!=i; j++) send(general[i], {mi_id, planes[j]});
p5: for (i = 0; i < N-1; i++)
     for (j = 0; j < N-1 AND j!=i; j++) receive(&id_aux, &planes_otros[i][j]);
p6: for (i = 0; i < N-1; i++)
      planes_finales[i] = MAYORÍA(planes[i], planes_otros[*][i]);
p7: plan_final = MAYORÍA(planes_finales, mi_plan);
```

¿es capaz de alcanzarse el consenso en presencia de fallos crash?

¿y en presencia de fallos bizantinos?

Algoritmo de los Generales Bizantinos (V)

- El algoritmo es correcto ante fallos de tipo *crash* (un *traidor* deja de enviar mensajes).
- Ante fallos *bizantinos* (un *traidor* envía mensajes arbitrarios), para que los *Generales leales* alcancen el consenso correcto es necesario:
  - Añadir una ronda de mensajes por cada *traidor.*
  - El número de *Generales leales* debe ser: 3t+1.

traidores	generales	mensajes
1	4	36
2	7	392
3	10	1790
4	13	5408

Algoritmo de los Generales Bizantinos (VI)

- Otros algoritmos:
  - *Flooding Algorithm:* Se reenvían *t+1* veces los *planes* recibidos. Sólo válido ante fallos *crash.*
  - *King Algorithm:* En cada ronda un *General* asume el rol de *Rey*, siendo su voto más importante que el del resto. Necesita *4t+1 leales* pero muchos menos mensajes.

traidores	generales	mensajes
1	5	48
2	9	240
3	13	672
4	17	1440