Sistemas Distribuidos - 21/22

Práctica 4: Raft 1ª parte

Autor: Unai Arronategui

Resumen

En la práctica 4 y 5, se plantea construir un servicio de almacenamiento clave/valor, en memoria RAM, tolerante a fallos utilizando replicación distribuida basada en Raft, una solución de máquina de estados replicada mediante un algoritmo de consenso. En el caso de esta práctica 4, se darán los primeros pasos para la operativa del algoritmo de elección de líder requerido, con algun que otro tipo de fallo, y el tratamiento, sin fallos, de añadir entradas (llamada rpc appendEntry) nuevas en las réplicas. Las referencias son el tema 8 de teoría y el documento adjunto al guión. La especificación de estado de las replicas y las llamadas RPC de referencia se encuentran disponibles en el anexo de este mismo guión.

Estas prácticas incluyen redactar una memoria, escribir código fuente y elaborar un juego de pruebas. El texto de la memoria y el código deben ser originales. Copiar supone un cero en la nota de prácticas.

Notas sobre esta práctica

- Aplicar "go fmt" al código fuente. Además : fijar en el editor máxima longitud de línea de 80 columnas, como mucho 20 instrucciones en una función (salvo situaciones especiales justificadas). Existen diferentes posibilidades de editores con coloración sintáctica, acceso a efinición de tipos variables y funciones, etc: vscode, gedit, geany, sublimetext, gvim, vim, ...
- La solución aportada deberá funcionar para diferentes pruebas.

1. Objetivo de la práctica

Los objetivos de la práctica son los siguientes:

- Presentar una implementación del algoritmo de eleccion de líder de Raft que funcione en escenarios iniciales de fallos básicos.
- Implementar el tratamiento de la llamada de RPC AppendEntry, suponiendo un funcionamiento sin fallos.

2. El servicio de elección de líder

Como primeros pasos en la solución completa de replicación distribuida, es necesario contruir el algoritmo de elección de líder por votacion explicado en clase de teoría y en el documento que se entrega asociado a este guión. En esta práctica se implementará, únicamente, la elección por votación mayoritaría (al menos la mitad más uno de las réplicas). La restricción, adicional, de nº de mandato y nº de índice para seleccionar al mejor líder (sección 5.4.1 del documento asociado) será implementado en la práctica 5.

2.1. Funcionamiento

Para la implementación, como comportamiento de alto nivel, seguir la especificación de la máquina de estados de la página 29 del tema 8 que especifica el comportamiento básico de cada nodo con respecto al algoritmo de elección de líder que deberá estar ejecutandose de forma continua. Por otra parte, la especificación má concreta de las llamadas RPC básicas y el estado de cada réplica se encuentra en la figura 2 disponible al final de este guión. En particular, el funcionamiento global de la elección del líder está diseminada en diferentes partes de esa figura.

Añadir el estado, definido en la figura 2, en el struct Raft definido en el fichero raft.go. También se necesitará otro struct que represente la información de cada entrada del registro de operaciones (logs) de Raft.Completar los structs RequestVoteArgs, para los argumentos de la llamada RPC RequestVote y RequestVoteReply para la respuesta de esa llamada.

Crear una gorutina concurrente que se responsabilice de la gestión del líder, y ponga en marcha un proceso de elección si no recibe mensajes de nadie durante un tiempo. De esta forma podrá saber quien es el líder, si ya lo hay, o convertirse el mismo en líder. Implementar el método asociado a la llamada RPC RequestVote() para solicitar votos en una elección.

Define los structs asociados a argumento y respuesta de la llamada RPC AppendEn-

tries(). Implementar el método asociado a la llamada RPC AppendEntries() para que el líder envíe látidos de corazón, de forma periódica, al resto de réplicas, notificando su buen estado operativo.

Aseguraros que los **tiempos de expiración**, asociados a la **recepción de latidos** y utilizados para la detección de fallo del líder, no expiren a la vez en diferentes réplicas (introducir variaciones aleatorias en rango limitado). Si no, los nodos votarán por si mismos y ninguno podrá convertirse en líder. Teneis disponible el paquete *rand* de la librería estandar para generar valores aleatorios en intervalos.

La frecuencia de látidos no debe ser superior a 20 veces por segundo. Y el periodo en el se decide un nuevo líder, trás detectar el fallo del anterior, no debe ser superior a 2,5 segundos. Se recuerda que puede requerirse varias elecciones para llegar a una decisión, por no conseguir mayorías. Luego hay que añadir un tiempo de expiración para una elección, en caso de no conseguir mayoría. Y debe ser suficientemente corto, relativo a al periodo de latido, para conseguir varias elecciones en el periodo de decisión de 2,5 segundos. Si hace falta utilizar intervalos aleatorios.

Apoyaros en el paquete *timer* de la líbrería estandar, tal como comentamos en la dispositiva 6 de Complementos Go,..."para gestionar acciones periódicas o trás retrasos.

EL RPC de Go solo envía structs con todos los campos exportados, es decir que empiezan en mayúsculas. Los campos de las substructuras internas (campos de la estructura de entrada del registro de operaciones de Raft) deben comenzar también con mayusculas. Si no, no funcionan las llamadas RPC.

3. Llamada RPC AppendEntry sin fallos

Implementar la llamada AppendEntry completa siguiendo las indicaciones de la figura 2 para conseguir enviarse y comprometer, con mayoría de réplicas, operaciones cliente en entradas de registro, sin aplicar las entradas comprometidas a la maquina de estados (eso se hará en práctica 5). Y probar dicha operativa, únicamente, durante funcionamiento sin fallos en el conjunto del sistema, incluido el líder.

4. Organización de código

Se aconseja el desarrollo del código y las pruebas en un ordenador Unix (Linux, BSDs, Mac, subsistema Linux de Windows). No se da soporte a desarrollo en Windows.

Implementar la funcionalidad de Raft, como algoritmo de consenso, en el fichero ïnternal/raft/raft.go", donde ya disponeis de un esqueleto.

La implementación del servicio Raft de consenso debe ofrecer el siguiente interfaz de llamadas y tipo dato:

```
// Crear nuevo nodo Raft
nd:= NuevoNodo(nodos, yo, canalAplicar)

// Someter operación para acuerdo por consenso en entrada de registro
nd.SometerOperacion(operacion interface{}) (indice, mandato,esLider)

// Obtención de estado de nodo Raft: quien es, mandato en curso
//y si cree ser el lider
nd.ObtenerEstado() (yo, mandato, esLider)

// cada vez que una nueva operacion es comprometida en una entrada
// de registro, cada nodo Raft debe enviar un mensaje AplicaOperacion
// a la máquina de estados
type AplicaOperacion
```

Teneis disponible la función "CallTimeout", de llamada a método remoto rpc con tiempo de expiración, en el fichero "internal/comun/rpctimeout".

Teneis un código básico de servidor rpc genérico (con tcp, no http) en fichero çmd/sr-vraft/main.go"para que lo adapteis al funcionamiento de servidor Raft con registro de lla-madas rpc AppendEntries y RequestVote desarrolladas en el fichero "internal/raft.raft.go".

Si os interesa, teneis disponible código de despliegue de programas remotos con s
sh en multiples máquinas en el fichero "internal/despliegue/sshClientWithPUBLICKEYAuthAndRemoteExec.go".

Desarrollar otro código ejecutable simple en "pkg/cltraft/cltraft.go" para enviar operaciones a los nodos Raft (a ser posible al lider) a través de llamadas rpc que definais y que espere su compromiso.

Teneis disponible código incompleto de test de integracion para las 4 pruebas de validacion en el fichero "internal/testintegracionraft1/testintegracionraft1.go".

Los nombres de directorios, en el camino de acceso a vuestro código Golang, no deben contener el carácter espacio ni otros caracteres que dificulten el acceso a los ficheros fuente para la ejecución de procesos remotos.

4.1. Validación

La mayor parte del desarrollo, se puede trabajar en la máquina local, pero para la validación final debe ejecutarse cada servidor en una máquina física diferente. Comprobar, previamente y con tiempo suficiente, que no hay problemas en ejecución distribuida.

Se plantean las siguientes pruebas a superar y desarrollar test especificos para cada una de ellas :

- 1. Arranque y parado de un nodo remoto
- 2. Se ha elegido a un primer líder correcto.
- 3. Un líder nuevo toma el relevo de uno caido.
- 4. Se consigue comprometer 3 operaciones seguidas en 3 entradas, con un líder estable y sin fallos en el sistema.

Teneis ejemplos de tests incompletos en el fichero "internal/raft/raft_test.go".

5. Evaluación

La realización de las prácticas es por parejas, pero los dos componentes de la pareja deberán entregarla de forma individual. En general, estos son los criterios de evaluación:

- Deben entregarse todos los programas, se valorará de forma negativa que falte algún programa / alguna funcionalidad.
- Los programas no tendrán problemas de compilación, se valorará de forma muy negativa que no compile algún programa.
- Todos los programas deben funcionar correctamente como se especifica en el problema través de la ejecución de la bateria de pruebas.
- Todos los programas tienen que seguir la guía de estilo de codificación de go fmt.
- Se valorará negativamente una inadecuada estructuración de la memoria, así como la inclusión de errores gramáticales u ortográficos.
- La memoria debería incluir diagramas de máquina de estados y diagramas de secuencia para explicar los **protocolos** de intercambio de mensajes y los eventos de fallo.
- Cada nodo(servidor) debe ejecutarse en una máquina física diferente en la prueba de evaluación.

La superación de las pruebas 1 y 2 supone la obtención de una B en la parte correspondiente a test. Para obtener una calificación de A, se deberá superar la prueba 1, 2 y 3. La superación de los test 1, 2, 3 y 4 supone tener una calificación de A+. Para llevar a cabo esta implementación, podeis basaros en el código disponible en el esqueleto.

5.1. Rúbrica

Con el objetivo de que, tanto los profesores como los estudiantes de esta asignatura por igual, puedan tener unos criterios de evaluación objetivos y justos, se propone la siguiente rúbrica en el Cuadro 1. Los valores de las celdas son los valores mínimos que hay que alcanzar para conseguir la calificación correspondiente y tienen el siguiente significado:

- A+ (excelente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea correctamente el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, sin errores. En el caso de la memoria, se valorará una estructura y una presentación adecuadas, la corrección del lenguaje así como el contenido explica de forma precisa los conceptos involucrados en la práctica. En el caso del código, este se ajusta exactamente a las guías de estilo propuestas.
- A (bueno). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. Plantea correctamente el problema a partir del enunciado propuesto e identifica las opciones para su resolución. Aplica el método de resolución adecuado e identifica la corrección de la solución, con ciertos errores no graves. Por ejemplo, algunos pequeños casos (marginales) no se contemplan o no funionan correctamente. En el caso del código, este se ajusta casi exactamente a las guías de estilo propuestas.
- B (suficiente). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. No plantea correctamente el problema a partir del enunciado propuesto y/o no identifica las opciones para su resolución. No aplica el método de resolución adecuado y / o identifica la corrección de la solución, pero con errores. En el caso de la memoria, bien la estructura y / o la presentación son mejorables, el lenguaje presenta deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, este se ajusta a las guías de estilo propuestas, pero es mejorable.
- B- (suficiente, con deficiencias). En el caso de software, conoce y utiliza de forma autónoma y correcta las herramientas, instrumentos y aplicativos software necesarios para el desarrollo de la práctica. No plantea correctamente el problema a partir del

enunciado propuesto y/o no identifica las opciones para su resolución. No se aplica el método de resolución adecuado y/o se identifica la corrección de la solución, pero con errores de cierta gravedad y/o sin proporcionar una solución completa. En el caso de la memoria, bien la estructura y / o la presentación son manifiestamente mejorables, el lenguaje presenta serias deficiencias y / o el contenido no explica de forma precisa los conceptos importantes involucrados en la práctica. En el caso del código, hay que mejorarlo para que se ajuste a las guías de estilo propuestas.

C (deficiente). El software no compila o presenta errores graves. La memoria no presenta una estructura coherente y/o el lenguaje utilizado es pobre y/o contiene errores gramaticales y/o ortográficos. En el caso del código, este no se ajusta exactamente a las guías de estilo propuestas.

Calificación	Sistema	Tests	Código	Memoria
10	A+	A+ (test 1-9)	A+	A+
9	A+	A+ (test 1-9)	A	A
8	A	A (test 1-7)	A	A
7	A	A (test 1-7)	В	В
6	В	B (test 1-5)	В	В
5	B-	B-(test 1-4)	B-	B-
suspenso		1 C		

Cuadro 1: Detalle de la rúbrica: los valores denotan valores mínimos que al menos se deben alcanzar para obtener la calificación correspondiente

6. Entrega y evaluación

Cada alumno debe entregar un solo fichero en formato tar.gz o zip, a través de moodle en la actividad habilitada a tal efecto, **no más tarde del día anterior** a la siguiente sesión de prácticas (b5).

La entrega DEBE contener los diferentes ficheros de código Golang y la memoria (con un máximo de 6 páginas la memoria principal y 10 más para anexos), en formato pdf. El **nombre del fichero tar.gz debe indicar apellidos del alumno y nº de práctica**. Aquellos alumnos que no entreguen la práctica no serán calificados. La evaluación "in situ" de la práctica se realizará durante la 5ª sesión de prácticas correspondiente.

State

Persistent state on all servers:

(Updated on stable storage before responding to RPCs)

latest term server has seen (initialized to 0 currentTerm

on first boot, increases monotonically)

votedFor candidateId that received vote in current

term (or null if none)

log[] log entries; each entry contains command

for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex index of highest log entry known to be

committed (initialized to 0, increases

monotonically)

lastApplied index of highest log entry applied to state

machine (initialized to 0, increases

monotonically)

Volatile state on leaders:

(Reinitialized after election)

nextIndex[] for each server, index of the next log entry

to send to that server (initialized to leader

last log index + 1)

for each server, index of highest log entry matchIndex[]

known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

leader's term term

leaderId so follower can redirect clients

prevLogIndex index of log entry immediately preceding

prevLogTerm term of prevLogIndex entry

log entries to store (empty for heartbeat; entries[]

may send more than one for efficiency)

leaderCommit leader's commitIndex

Results:

currentTerm, for leader to update itself term true if follower contained entry matching success

prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)

- 2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
- 3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
- Append any new entries not already in the log
- If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

candidate's term term candidate requesting vote candidateId

lastLogIndex index of candidate's last log entry (§5.4) lastLogTerm term of candidate's last log entry (§5.4)

Results:

term currentTerm, for candidate to update itself voteGranted true means candidate received vote

Receiver implementation:

- 1. Reply false if term < currentTerm (§5.1)
- If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- · On conversion to candidate, start election:
 - Increment currentTerm
 - Vote for self
 - Reset election timer
 - · Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- · If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- · If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
 - If successful: update nextIndex and matchIndex for follower (§5.3)
- If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] \geq N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [31] describes the algorithm more precisely.