

30221/39521 - SISTEMAS DISTRIBUIDOS
Universidad de Zaragoza, curso 2023/2024

Práctica 4: Raft 2ª parte

Resumen

En la práctica 4, 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. La llamada *rpc raft.SometerOperacion*, enviada al líder, distribuye la operación al resto de réplicas mediante invocaciones RPC *AppendEntries*. Las referencias son el tema 5 de teoría y el documento original de Raft. La especificación de estado de las réplicas y las llamadas RPC de referencia se encuentran disponibles en el anexo al final 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 efinition de tipos variables y funciones, etc: vscode, kate, gedit, geany, sublimetext, gvim, vim, emacs ...
- 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 completo que funcione en escenarios de fallos variados.

- Implementar el tratamiento de la llamada de RPC *AppendEntry*, suponiendo un funcionamiento CON fallos.

2. Eleccion de líder completa y llamada RPC *AppendEntries* con fallos

Se plantea implementar la solución completa de elección de líder, incluida la restricción de nº de mandato y nº de índice para seleccionar al mejor líder (sección 5.4.1 del documento raft original), y la operativa de llamada RPC *SometerOperacion* por parte de los clientes del sistema replicado que provocará llamadas *AppendEntries* desde el líder a los seguidores con avance del índice de entradas de registro comprometidas. Ambos se desarrollarán en diferentes escenarios de fallos. Además se aplicará las operaciones sobre una máquina de estados simple, con operaciones de lectura y escritura sobre un almacén de datos en RAM mediante un tipo de datos *map* de Golang.

2.1. Funcionamiento

Aunque sea el líder el único que inicia la operativa de añadir nuevas entradas en el registro, todas las replicas tienen que trasladar, de forma independiente, las operaciones comprometidas al servicio local de almacenamiento (máquina de estados) a través del canal de aplicación de operación. Mantener ambas actividades bien separadas y tener cuidado de no incurrir en problemas de exclusión mutua.

3. 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 "raft/internal/raft/raft.go", donde ya disponeis de un esqueleto.

El conjunto del código reside en el *modulo raft*, bajo el cual encontrais diferentes paquetes y funcionalidades, en subdirectorios. El subdirectorio "raft/cmd" es utilizado para ubicar código ejecutable (func main). El subdirectorio "raft/internal" se utiliza para paquetes de uso interno al modulo. En el subdirectorio "raft/pkg" se ubican paquetes a exportar, que pueden ser utilizados por código externo al modulo. Y el subdirectorio "raft/vendor" contiene los paquetes importados por el código de este modulo, y es obtenido

Práctica 4: Raft 2ª parte

ejecutando **"go mod vendor"** en el directorio raíz del modulo.

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

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

// Someter operación para acuerdo por consenso en entrada de registro
nr.SometerOperacion(operacion Operacion) (indice, mandato, esLider, idLider, valorADevolver)

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

// Metodo Para() utilizado cuando no se necesita mas al nodo
func (nr *NodoRaft) Para()

// 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 *"raft/internal/comun/rpctimeout"*. Se ha actualizado con nueva versión que abre y cierra las conexiones TCP para cada llamada.

Teneis un código básico de servidor rpc genérico (con tcp, no http) en fichero *"cmd/sr-raft/main.go"* (ya operativo) para que lo adapteis al funcionamiento de servidor Raft con registro de llamadas rpc *AppendEntries* y *RequestVote* desarrolladas en el fichero *"internal/raft.raft.go"*. **En est fichero, además se especifica un TipoOperador nuevo que dispone, como únicas operaciones posibles, las operaciones de lectura (operación "leer") y escritura (operación "escribir") a comprometer y aplicar a la máquina de estados que puede consistir en un almacen de datos implementado en GO con un tipo *map[string]string*.**

Teneis disponible código incompleto de test de integracion para las 3 pruebas de validacion en el fichero *"raft/internal/testintegracionraft1/testintegracionraft1.go"*. Funciona con el primer test (mediante *"go test -v ./..."*), utilizando el ejecutable *"main.go"* disponible y el código de despliegue *"sshClientWithPUBLICKEYAuthAndRemoteExec.go"*. En el uso de ssh, los tests de integración disponibles utilizan el algoritmo de clave pública *"ed25519"* (línea 44, fichero *"testintegracionraft1.go"*, indica fichero de clave privada).

Práctica 4: Raft 2ª parte

Podeis adaptarlo, sea actualizandolo al fichero de clave privada, de ssh, que teneis habilitado en vuestra cuenta de usuario, sea generando una pareja nueva de claves público/privada de tipo ed25519 mediante `"ssh-keygen -t ed25519"`. Además, los test de integración presuponen una ubicación determinada del camino de acceso al directorio del modulo de la práctica (línea 48, fichero `"testintegracionraft1.go"`). Modificarlo para indicar vuestro propio camino de acceso.

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.

3.1. 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 específicos para cada una de ellas :

1. Se consigue acuerdos de varias entradas de registro a pesar de que un replica (de un grupo Raft de 3) se desconecta del grupo
2. NO se consigue acuerdo de varias entradas al desconectarse 2 nodos Raft de 3.
3. Someter 5 operaciones cliente de forma concurrente y comprobar avance de índice del registro.

Teneis ejemplos de tests incompletos en el fichero `"internal/raft/integracionraft1.go"`.

Para ejecutar todos los tests del modulo podeis ejecutar `"go test ./..."` en el directorio raíz del modulo. Hay disponibles tambien métodos para ejecutar tests específicos. El comando `"go test -v ./..."` os permite tener más salida de información de depuración en la ejecución de tests completos de un modulo.

4. 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:

Práctica 4: Raft 2ª parte

- 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 batería 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 gramaticales 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 réplica debe ejecutarse en una máquina física diferente en la prueba de evaluación.**

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

4.1. 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

Práctica 4: Raft 2ª parte

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 funcionan 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.

Práctica 4: Raft 2ª parte

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)	B	B
6	B	B (test 1-5)	B	B
5	B-	B-(test 1-4)	B-	B-
suspense	1 C			

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

5. 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.

Práctica 4: Raft 2ª parte

State	RequestVote RPC
<p>Persistent state on all servers: (Updated on stable storage before responding to RPCs)</p> <p>currentTerm latest term server has seen (initialized to 0 on first boot, increases monotonically)</p> <p>votedFor candidateId that received vote in current term (or null if none)</p> <p>log[] log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)</p> <p>Volatile state on all servers:</p> <p>commitIndex index of highest log entry known to be committed (initialized to 0, increases monotonically)</p> <p>lastApplied index of highest log entry applied to state machine (initialized to 0, increases monotonically)</p> <p>Volatile state on leaders: (Reinitialized after election)</p> <p>nextIndex[] for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)</p> <p>matchIndex[] for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)</p>	<p>Invoked by candidates to gather votes (§5.2).</p> <p>Arguments:</p> <p>term candidate's term</p> <p>candidateId candidate requesting vote</p> <p>lastLogIndex index of candidate's last log entry (§5.4)</p> <p>lastLogTerm term of candidate's last log entry (§5.4)</p> <p>Results:</p> <p>term currentTerm, for candidate to update itself</p> <p>voteGranted true means candidate received vote</p> <p>Receiver implementation:</p> <ol style="list-style-type: none"> 1. Reply false if term < currentTerm (§5.1) 2. 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)
AppendEntries RPC	Rules for Servers
<p>Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).</p> <p>Arguments:</p> <p>term leader's term</p> <p>leaderId so follower can redirect clients</p> <p>prevLogIndex index of log entry immediately preceding new ones</p> <p>prevLogTerm term of prevLogIndex entry</p> <p>entries[] log entries to store (empty for heartbeat; may send more than one for efficiency)</p> <p>leaderCommit leader's commitIndex</p> <p>Results:</p> <p>term currentTerm, for leader to update itself</p> <p>success true if follower contained entry matching prevLogIndex and prevLogTerm</p> <p>Receiver implementation:</p> <ol style="list-style-type: none"> 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) 4. Append any new entries not already in the log 5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry) 	<p>All Servers:</p> <ul style="list-style-type: none"> • 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) <p>Followers (§5.2):</p> <ul style="list-style-type: none"> • 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 <p>Candidates (§5.2):</p> <ul style="list-style-type: none"> • On conversion to candidate, start election: <ul style="list-style-type: none"> • 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 <p>Leaders:</p> <ul style="list-style-type: none"> • 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 <ul style="list-style-type: none"> • 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] ≥ 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.