Realizzare servizi Byzantine Fault-Tolerant e Fault-Scalable: **protocollo Query/Update** (Q/U)

Elaborato in SISTEMI DISTRIBUITI

Daniele Giulianini

Anno accademico 2019-2020

Tolleranza ai guasti: modello di **sistema** (distribuito)

- **Modellazione** svolge ruolo centrale nell'elaborazione di soluzioni alla **tolleranza ai guasti**
- Diverse prospettive:
  - Grado di sincronia
  - Topologia della rete
  - …
- Modelli di sistema per grado di sincronia:
  - sincroni
  - parzialmente sincroni
  - asincroni

Tolleranza ai guasti: modello di **guasto**

- In letteratura numerosi
- Esempi, in ordine di severità:
  - Crash
  - Crash-recovery
  - Receive-omission
  - **Byzantine**
    - Byzantine Faut Tolerance (BFT)
- Influenzano il # di repliche necessarie a tollerare il tipo di guasti scelti
- Si possono classificare anche modelli per i canali di comunicazione
- Può anche considerarsi modello ibrido

Tolleranza ai guasti bizantini

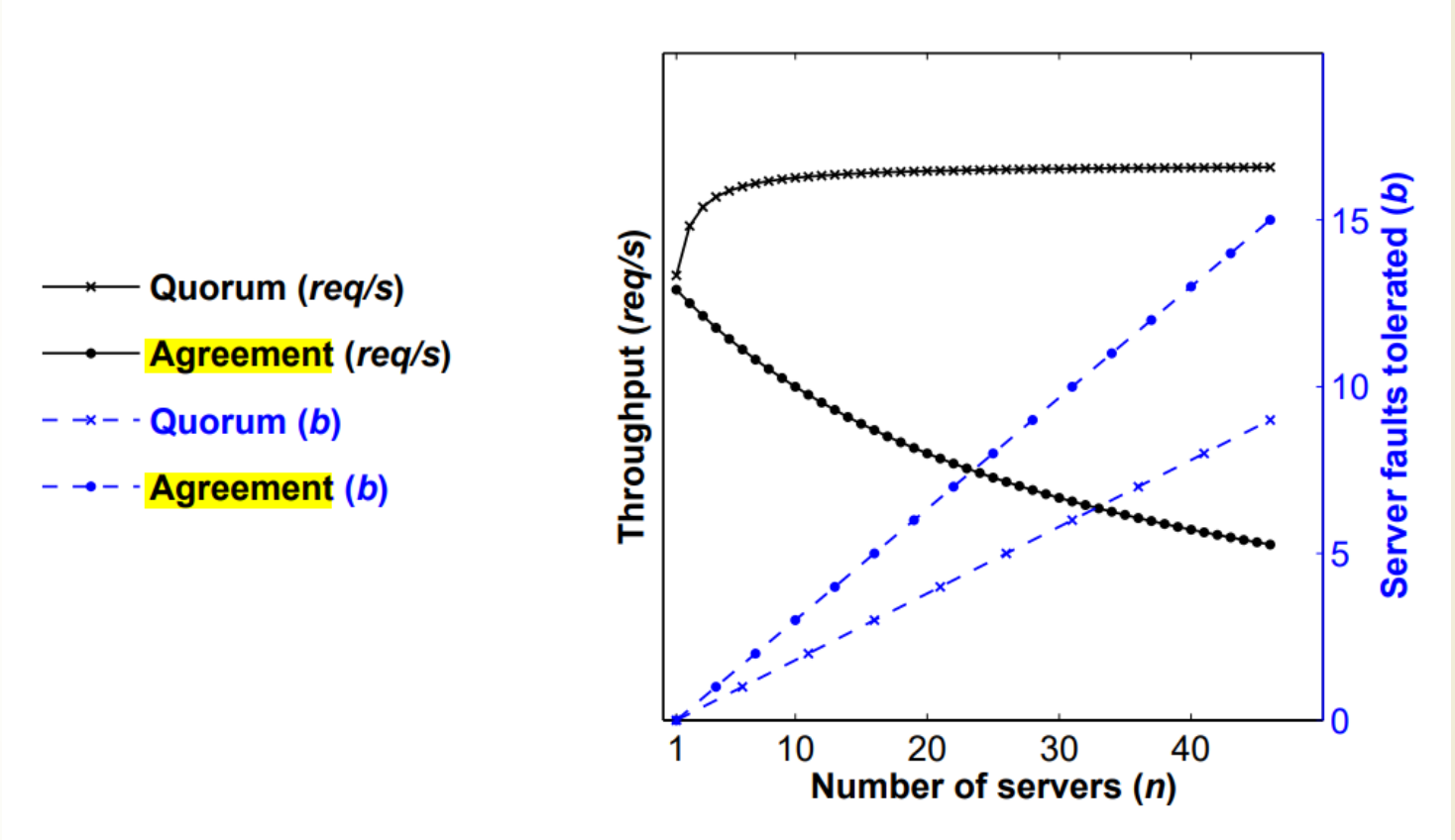
- Soluzioni in letteratura numerose e variegate
  - Approcci agreement-based e quorum-based
- Rinnovato interesse con l'avvento della **blockchain**
- No **one-size-fits-all** [1, 2, 3, 4, 5]
  - Ognuno eccelle per configurazioni operative differenti
  - Perfino all'interno dello stesso protocollo esse impattano pesantemente le prestazioni
  - La scelta del più adeguato è materia **ingegneristica**

State Machine Replication

- Approccio consolidato per soluzioni BFT
  - Introdotto da Lamport
  - Si modella un servizio come una **macchina a stati deterministica**
- Repliche:
  - iniziano nello stesso stato iniziale
  - Eseguono i comandi nello stesso ordine
  - => stesso stato finale
- Client:
  - Considerano le risposte quali voti per la risposta «corretta» [6]
- Requisito: **consenso** sulla sequenza ordinata di comandi (**Atomic Broadcast**)
  - Che è il reale apporto di ognuno di questi protocolli
- Un limite: determinismo è un requisito per il servizio
    - Affrontato da alcune nuove tendenze (Sieve[7] e Mastercrypt[7])

Fault-scalability

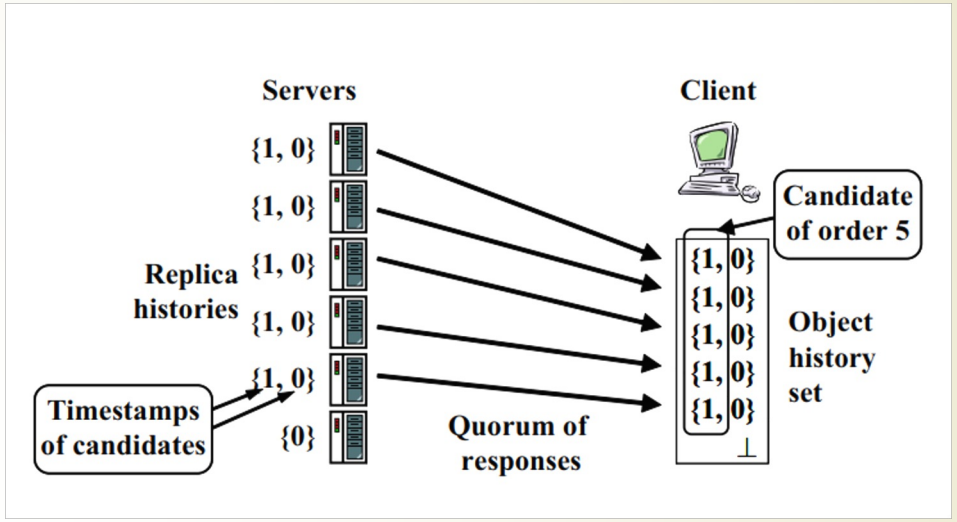
- capacità di garantire buone prestazioni, con un degradamento graduale, e non repentino, nell'erogazione di un servizio all'aumentare dei **guasti** dei server che lo erogano [8]



Q/U: panoramica

- Protocollo BFT client/server
- basato su quorum (wrt agreement)
- API operations-based (SMR)
- Servizi fault-scalable
- richiede **5b + 1** repliche per tollerare **b** guasti bizantini
- protocollo base («single-object update») + ottimizzazioni + estensione «multi-object update»
- Principale caratteristica: approccio **ottimistico**
- Modello:
  - Di sistema: parzialmente sincrono
  - Di guasto: crash-recovery + byzantine (client/server)

Q/U: concetti chiave - I



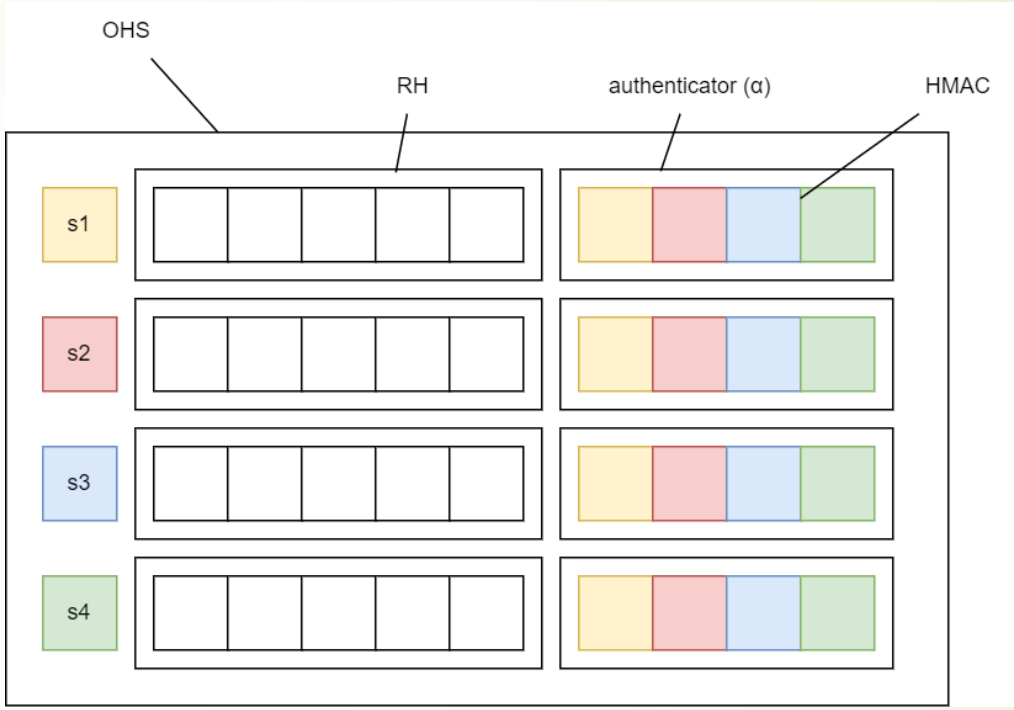
- 2 soli ruoli: client (autenticati) e replica
- 2 tipi di operazioni (deterministiche):
  - query e update
- Replica:
  - riceve richieste dei client
  - invoca (se accettate) un *metodo* che determina (se update) nuova versione dell'oggetto replicato
    - Identificata da **Logical Timestamp** (**LT**)
  - Conserva versioni oggetto in **Replica History** (**RH**) composta da **LT**
- Client:
  - Esegue operazioni inviando *richieste*
  - Conserva le RH ritornate dalle repliche in **Object History Set** (**OHS**)

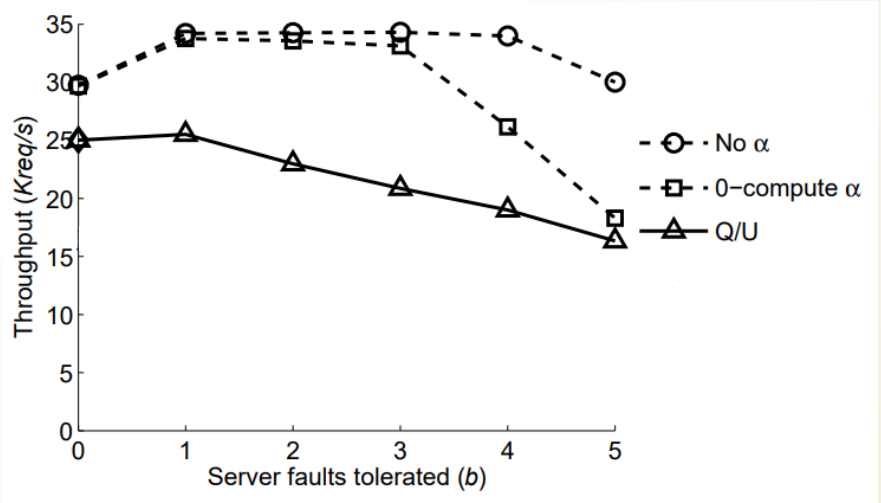
Q/U: concetti chiave – II

- OHS sottoposta a **classificazione**:
  - Se OHS **non** evidenzia **guasti** o **contesa:**
    - esecuzione operazione in singolo passo di comunicazione (approccio «ottimistico»)
  - Se OHS evidenzia **contesa:**
    - procedura di recupero (**repair**) basata su **backoff** (random exponential) client e # ritrasmissioni con timeout della richiesta
- Classificazione:
  - basata su sistema di quorum a **soglie**
  - viene ricondotta all'**ordine** del candidato
  - Relazione contiene il dettaglio delle assunzioni matematiche

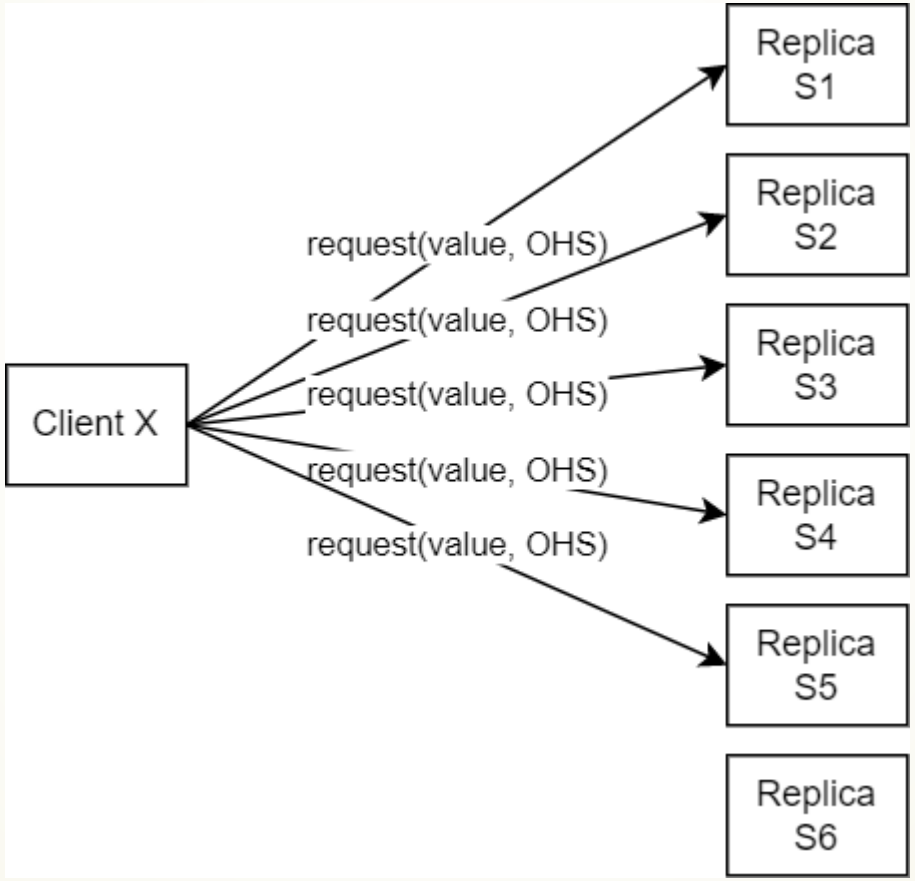
Verifica integrità Replica History

- Rende statisticamente improbabile corruzione inosservata delle RH
- Basato su "Hash-based Message Authentication Code" (**HMAC**) [9]
- Ogni server possiede, per ogni altro server, una **chiave segreta** con esso condivisa
- *Digital signature* come alternativa

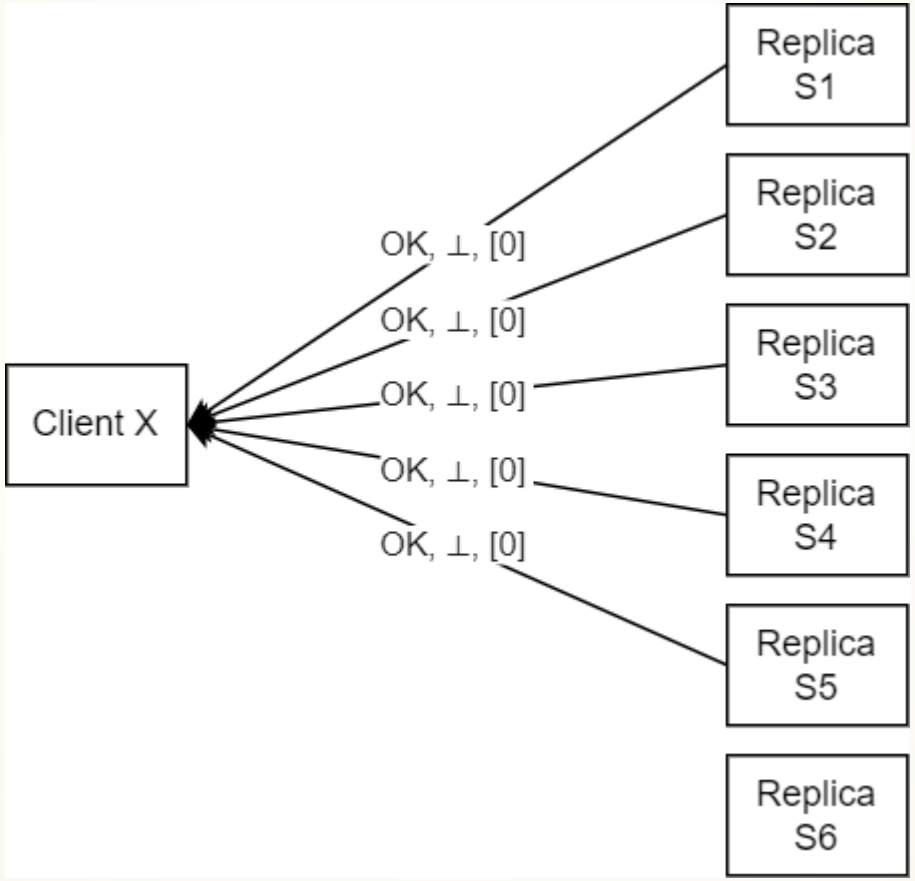




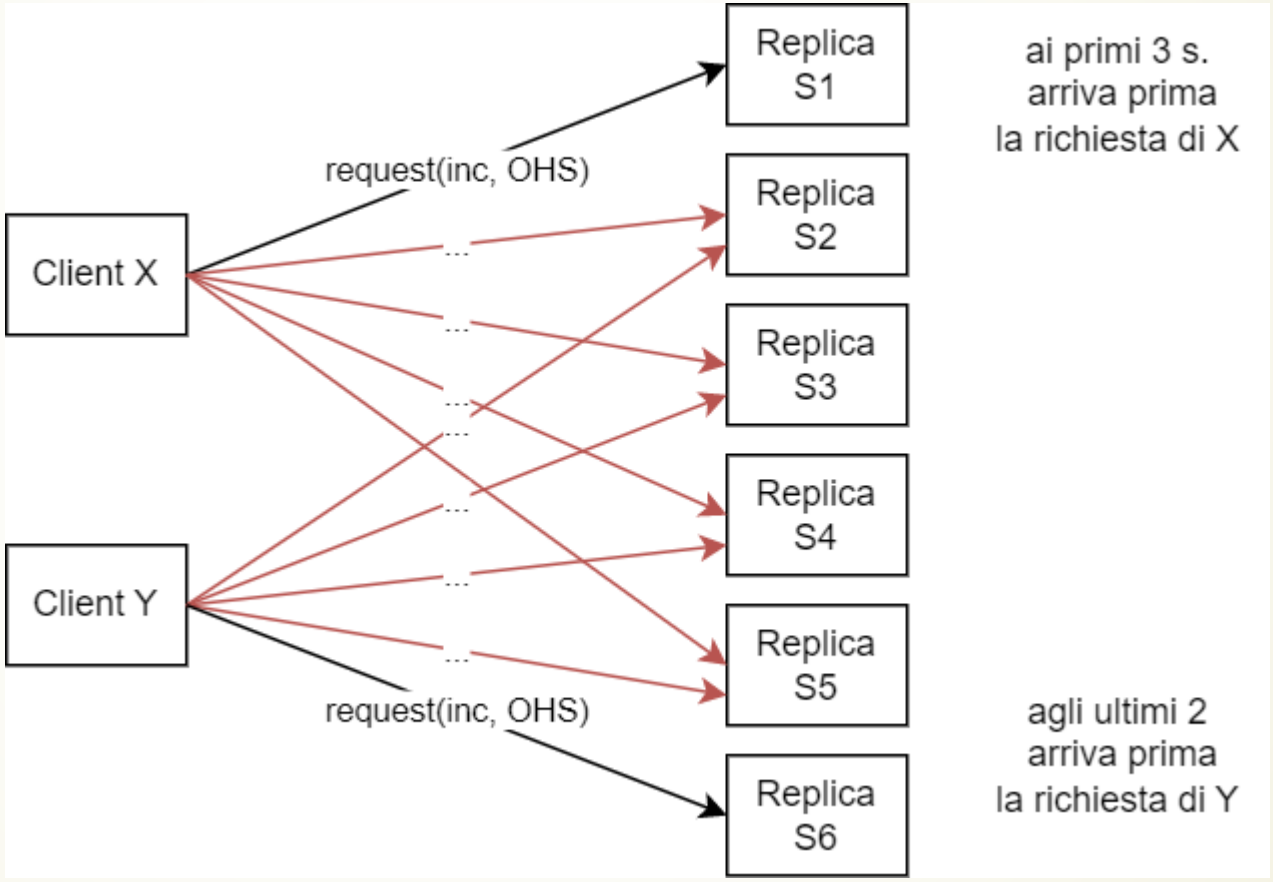
Scenario di non contesa (ottimistico) - I



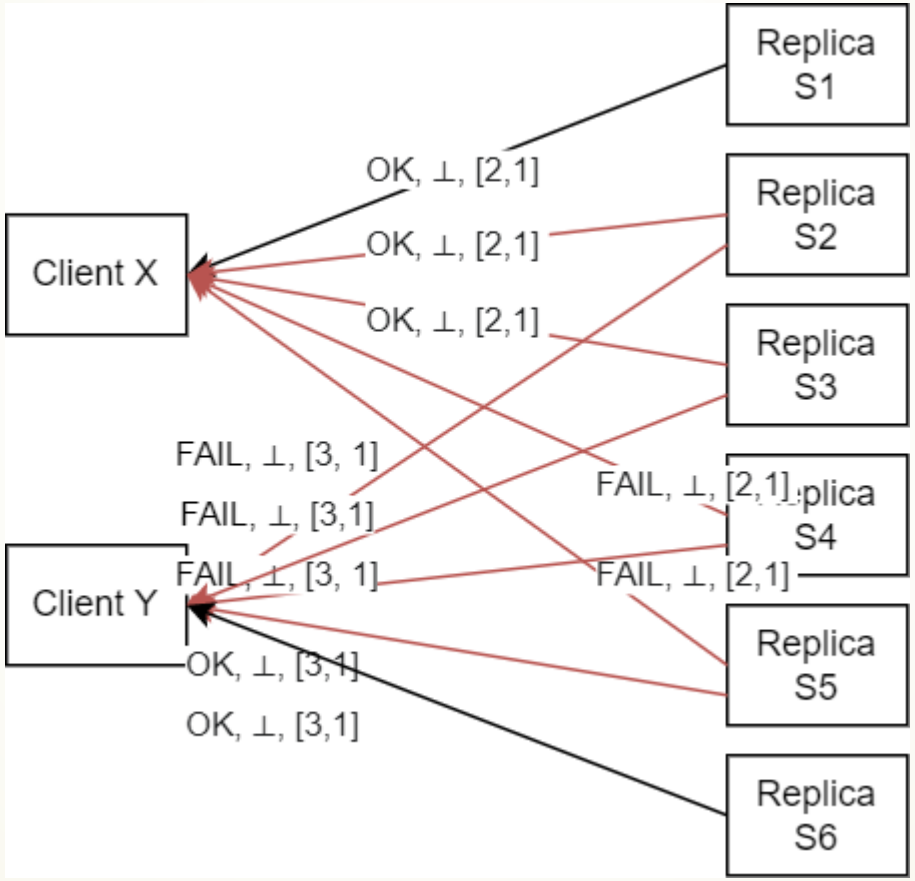
Scenario di non contesa (ottimistico) - II



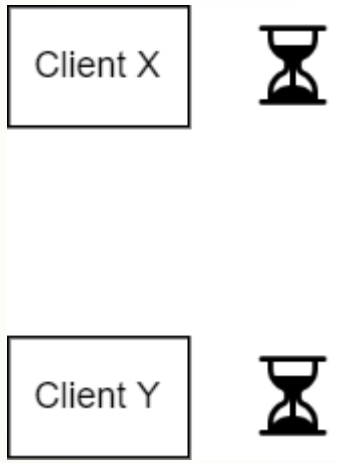
Scenario di contesa - I



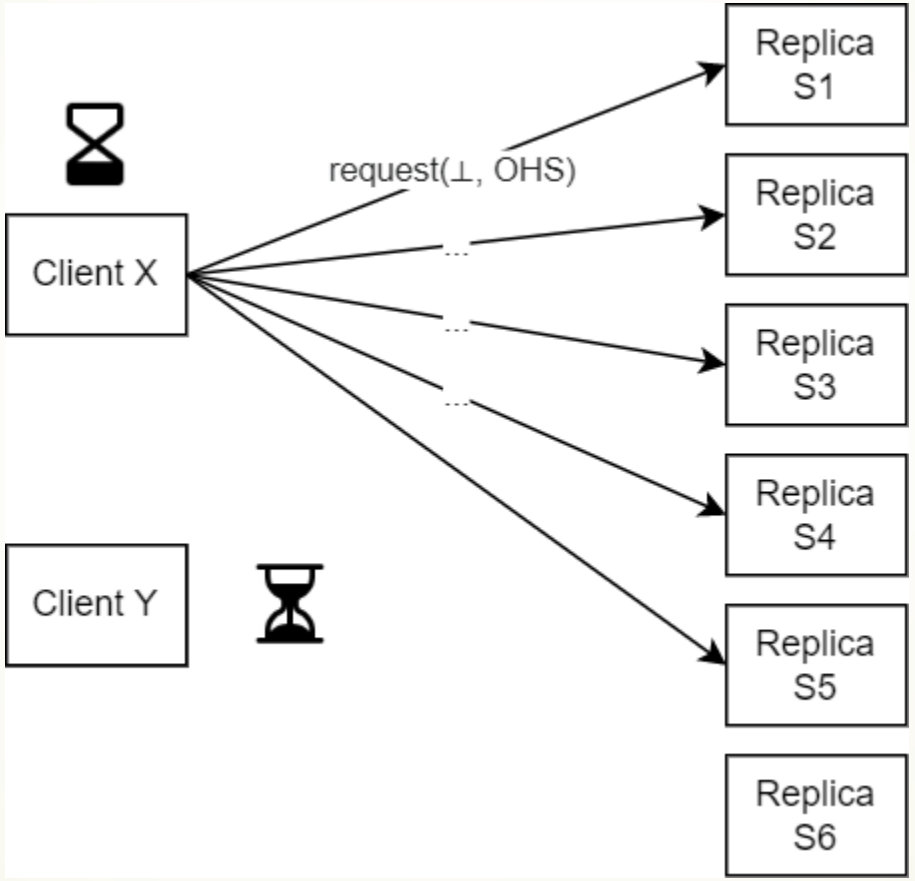
Scenario di contesa - II



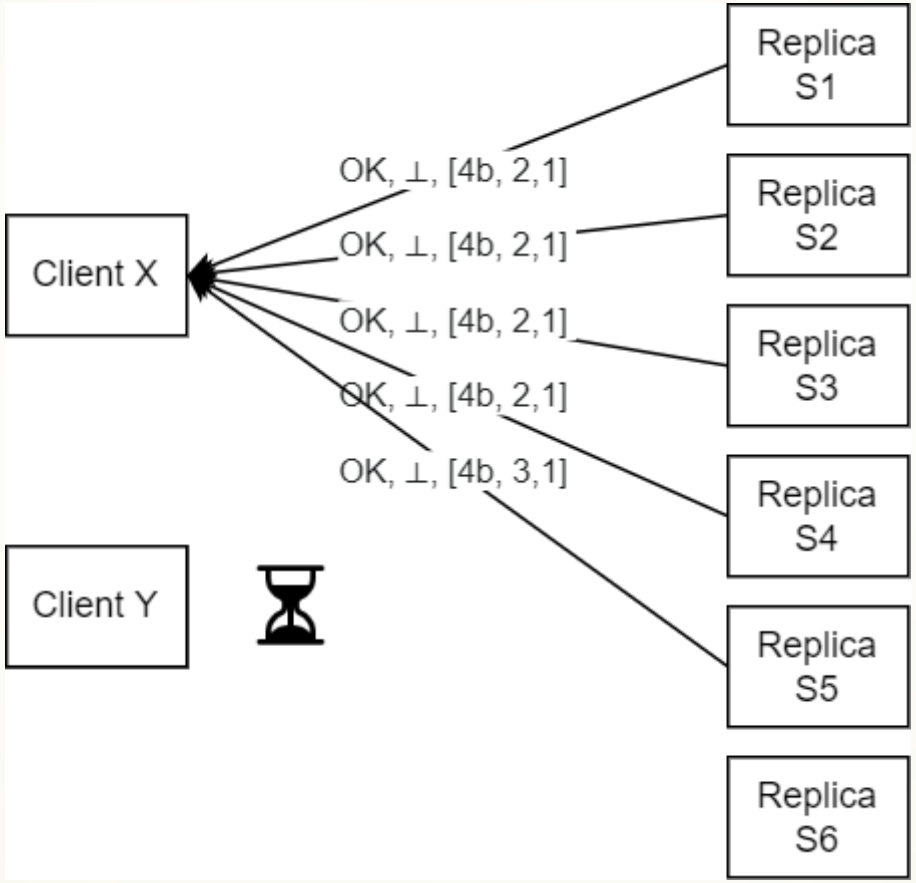
Scenario di contesa – III



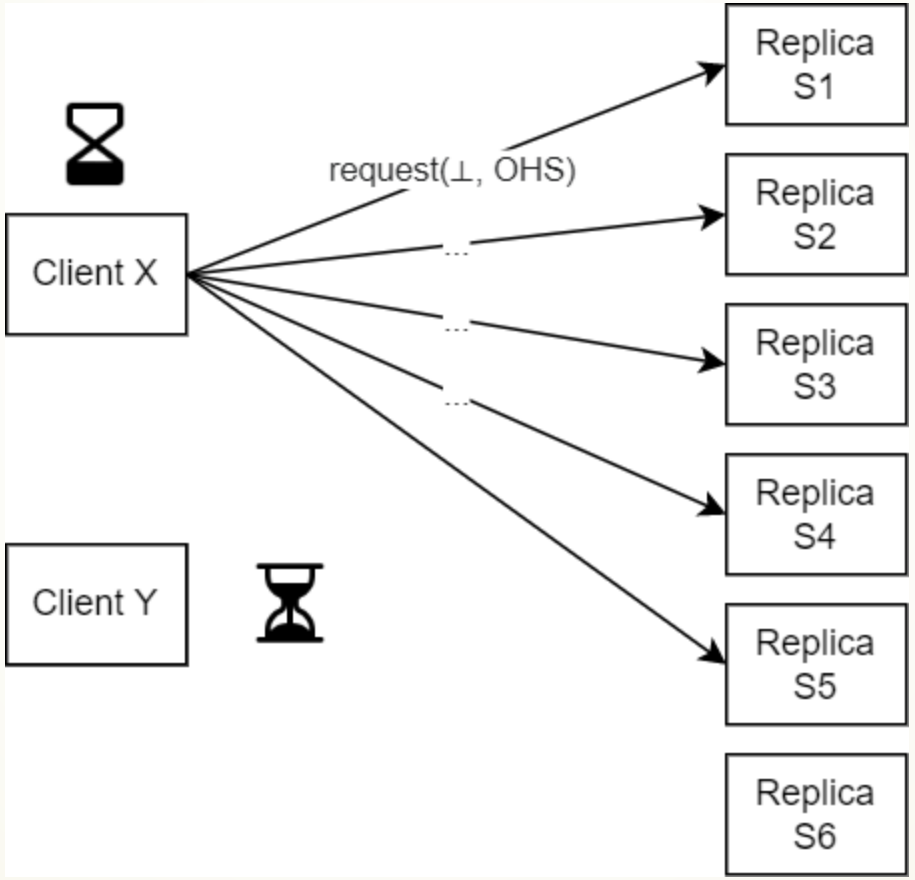
Scenario di contesa - IV



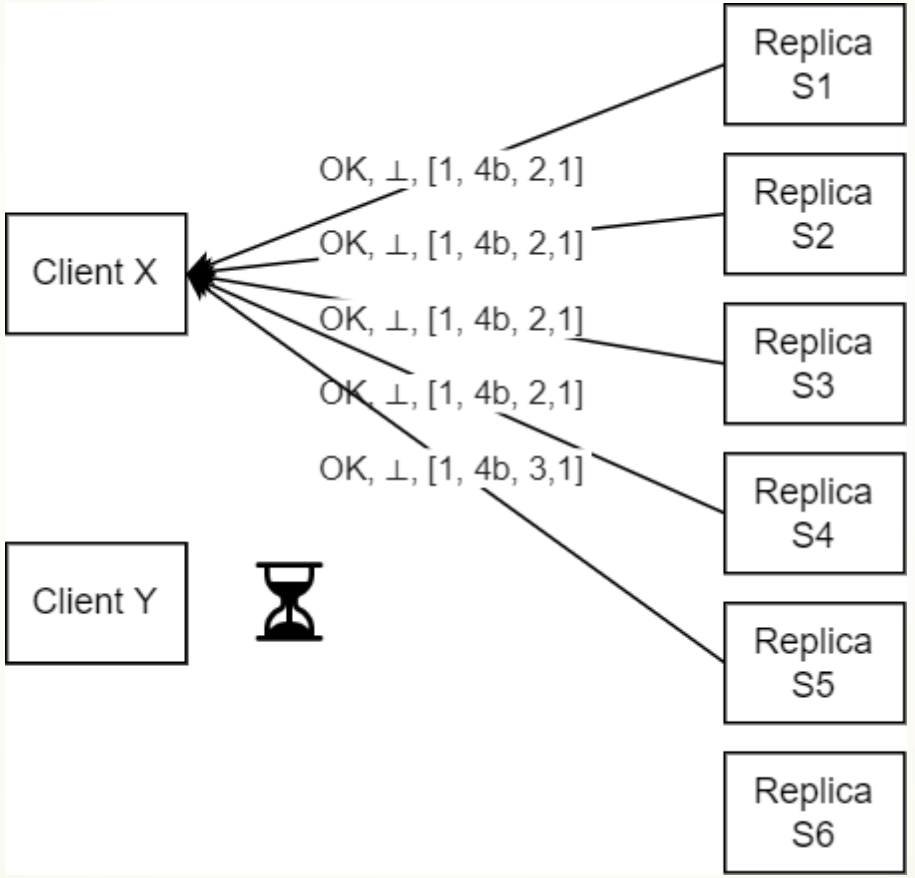
Scenario di contesa - V



Scenario di contesa - VI



Scenario di contesa - VII



Altri scenari

- Guasto crash-stop a replica
- Object-sync + inline-repair
  - => dettaglio nella relazione

Proprietà e garanzie

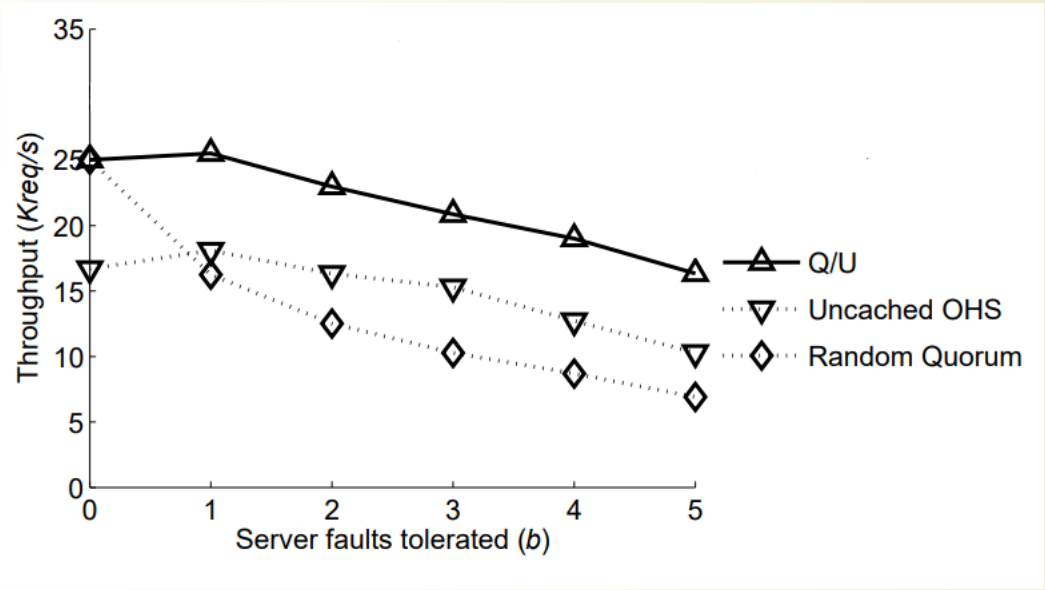
- Linearizability (single-object) e Strict Serializability (multi-object)
- Fault-scalability
- Failure atomicity
- Concurrency atomicity
- Liveness
  - Se esecuzione maligne => decade, ma mitigazione efficace
- Throughput scalability
  - Scaleup pitfall => multi-object uipdate

Ottimizzazioni ed estensioni

- Cached OHS (si veda grafico a lato: ~55/70%)
- Quorum preferito (si veda grafico a lato: ~40/60%)
- Inline repair
  - "Scorciatoia" al repair per candidati particolari, detti "riparabili"
- Query ottimistica
- Gestione richieste ripetute
- Multi-object update
  - Aggiornamento atomico di più oggetti
  - OHS => multi-object OHS
  - Da linearizability a strict ser.
- Timestamp compatto
- Pruning Replica History
- Mitigazione starvation con guasti bizantini

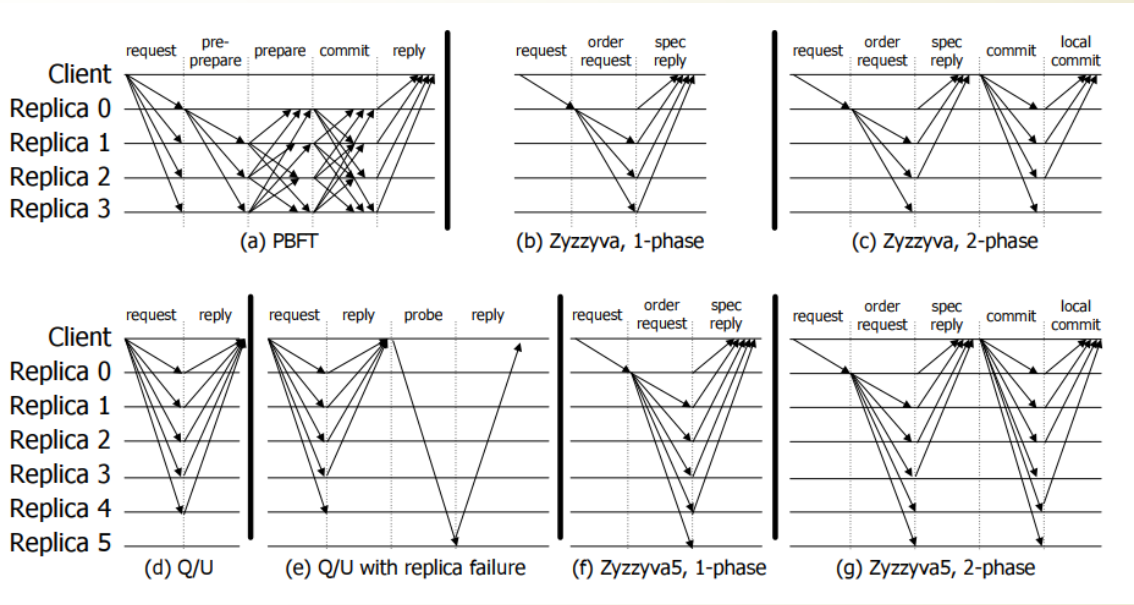
Incentivazione approccio ottimistico

Riduzione banda e occupazione



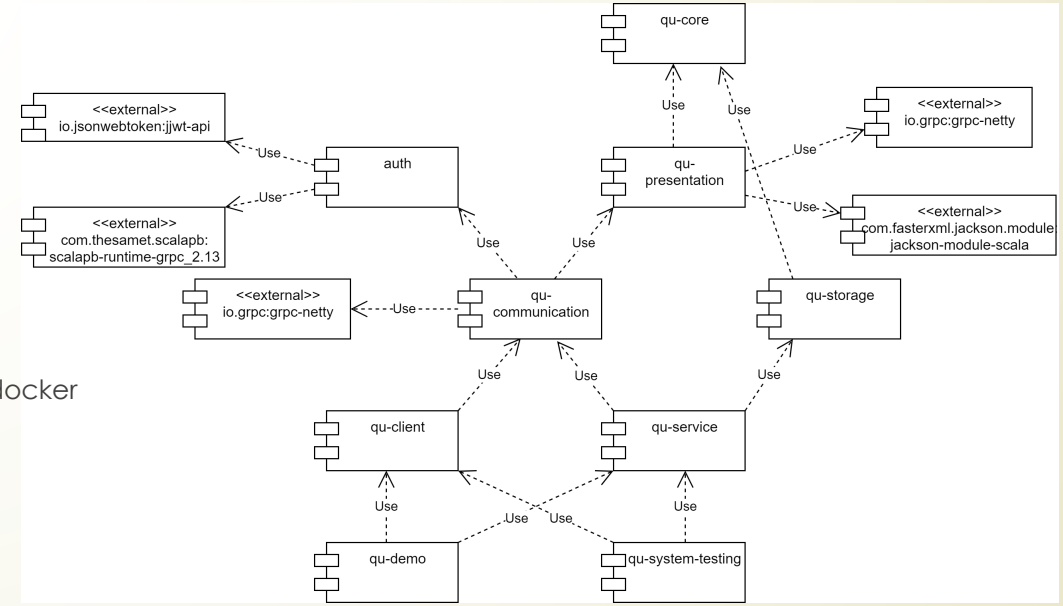
Bilancio Q/U

- Pro:
  - Ottime prestazioni in assenza di contesa e/o guasto [inline] «riparabile»
    - In particolare, all'aumentare di:
      - dim. del payload
      - Latenza replica-replica [3]
- Contro:
  - Procedura di repair (guasti non «riparabili» e/o contesa) onerosa
  - Il servizio deve essere deterministico



Progetto

- Premesse:
  - No **one-size-fits-all**
  - assenza di una implementazione di Q/U disponibile in rete
- **Rivisitazione asincrona** e allo stato dell'arte di Q/U
  - scala, gRPC, Protobuf ...
- Focus prestabiliti:
  - Generalità e riusabilità per l'utente programmatore
  - Modularità, estensibilità e compatibilità con differenti tecnologie di:
    - Serializzazione
    - Autenticazione
    - Comunicazione
  - Type safety con generici (type system di scala)
- Deliverable:
  - Libreria + insieme di specifiche + Demo
- Workflow di sviluppo
  - Gitlab flow
  - continuous integration/continuous delivery (CI/CD) con docker
  - build automation con sbt



Validazione

- **Testing automatizzato** come principale forma di validazione
  - Demo come secondaria
- Test d'unità e di sistema
- Approccio:
  - Behaviour-driven-development
    - Test sono in realtà **specifiche di funzionamento**
  - Testing efficace
  - Property-based testing

```
describe( description = "A ConcreteLogicalTimestamp") {

  //test ordering
  describe( description = "when initialized with a logical time") {
    it("should be classified as previous to a ConcreteLogicalTimestamp " +
      "with a greater logical time") {
      forAll(arbitraryLt) { aLt =>
        forAll(logicalTimestampWithGreaterTimeThan(aLt.time)) { lt =>
          lt should be > aLt
        }
      }
    }
  }
}
```

```
describe( description = "A client") {
  val queryOp = GetObj[Int]()

  describe( description = "when requesting a query operation and receiving" +
    " a response with order >= q " +
    "and an ohs with method") {

    it("should return the correct answer in a single round of communication") {
      val expectedResponse = initialValue + 1

      queryQuorum.expects(Some(queryOp), emptyOhs(serversIdsAsSet), *, *)
        .noMoreThanOnce()
        .returning(Future.successful(
          (Some(expectedResponse), thresholds.q, ohsWithMethodFor(serversKeys))))
      (mockedBackOffPolicy.backOff()(_: ExecutionContext)).expects(*).never()

      whenReady(client.submit[Int](queryOp)) {
        _ should be(expectedResponse)
      }
    }
  }
}
```

```
describe( description = "when multiple updates are issued followed by a query") {

  it("should return to client the correct answer") {
    val nIncrements = 3
    val operations = List.fill(nIncrements)(Increment())
    for {
      authenticatedQuClient <- quClientAsFuture
      _ <- seqFutures(operations)(authenticatedQuClient.submit(_))
      queryResult <- authenticatedQuClient.submit(GetObj())
    } yield queryResult should be(InitialObject + nIncrements)

  }
}
```

Esempio d'uso: la libreria

- 3 stili di utilizzo (a diverso livello di astrazione):
  - gRPC-agnostic (livello a. intermedio)
  - Strutture dati pronte all'uso (livello a. più alto)
  - gRPC-aware (livello a. più basso, solo server-side)
- Un estratto, client-side… →

- Strutture dati pronte all'uso
  - Configurazione client Q/U:

```
val quServersInfo = Set(quServer1SocketAddr,
  quServer2SocketAddr /*, ...*/ ,
  quServer6SocketAddr)
val distributedCounter = DistributedCounter("username",
  "password", authServerSocketAddr.ip,
  authServerSocketAddr.port, quServersInfo, thresholds)
```

  - Sottomissione operazioni:

```
for {
  _ <- distributedCounter.incrementAsync()
  _ <- distributedCounter.resetAsync()
  _ <- distributedCounter.incrementAsync()
  _ <- distributedCounter.incrementAsync()
  _ <- distributedCounter.decrementAsync()
  value <- distributedCounter.valueAsync
} yield println("distributed counter value is:" + value)
```

- gRPC-agnostic
  - Configurazione client Q/U:

```
val authClient = AuthenticatingClient[Int](
  authServerSocketAddr.ip,
  authServerSocketAddr.port,
  username = "username",
  password = "password")
val authenticatedQuClient =
  for {
    builder <- for {
      _ <- authClient.register()
      builder <- authClient.authorize()
    } yield builder
  }
  yield builder.addServer(quServer1SocketAddr)
    //...
    .addServer(quServer6SocketAddr)
    .withThresholds(thresholds).build
```

  - Sottomissione operazioni:

```
for {
  authenticatedQuClient <- authenticatedQuClient
  _ <- authenticatedQuClient.submit[Unit](Increment())
  _ <- authenticatedQuClient.submit[Unit](Reset())
  _ <- authenticatedQuClient.submit[Unit](Increment())
  _ <- authenticatedQuClient.submit[Unit](Increment())
  _ <- authenticatedQuClient.submit[Unit](Decrement())
  value <- authenticatedQuClient.submit[Int](Value)
} yield println("distributed counter value is:" + value)
```

Sviluppi futuri

- Suggeriti dagli autori:
  - Politica di scelta quorum più efficiente
  - Ottimizzazione “update-query contention”
  - Supporto alla memorizzazione più efficiente e durabile
  - Estensione ”Multi-object update”
  - Adozione di meccanismi crittografici più efficienti per la autenticazione dei messaggi
  - Approcci di mitigazione al fallimento bizantino
- Ulteriori:
  - Servizio di distribuzione chiavi di sessione private
  - Introdurre tecnologia di serializzazione differente
  - Introdurre un meccanismo di autenticazione a livello di trasporto
  - Introdurre un meccanismo di autenticazione/autorizzazione più evoluto
  - Accesso alle funzionalità del protocollo attraverso API REST
  - Gestione credenziali più accurata
  - Completare le implementazioni per le strutture dati distribuite
  - Abilitare partecipazione dinamica al cluster di repliche

Bibliografia

1. James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In In Proc. OSDI, pages 177–190, 2006.
2. Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In Proceedings of twentyfirst ACM SIGOPS symposium on Operating systems principles, pages 45–58, 2007.
3. Singh, Atul, et al. "BFT Protocols Under Fire." NSDI. Vol. 8. 2008.
4. Rachid Guerraoui, Nikola Kne˘zevi´c, Vivien Qu´ema, and Marko Vukoli´c. The next 700 bft protocols. In Proceedings of the 5th European conference on Computer systems, pages 363–376, 2010.
5. Duan, Sisi, Michael K. Reiter, and Haibin Zhang. "BEAT: Asynchronous BFT made practical." Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. 2018.
6. Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys (CSUR), 22(4):299–319, 1990.
7. Christian Cachin, Simon Schubert, and Marko Vukoli´c. Non-determinism in byzantine fault-tolerant replication. arXiv preprint arXiv:1603.07351, 2016.
8. Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. ACM SIGOPS Operating Systems Review, 39(5):59–74, 2005.
9. Bellare, Mihir, Ran Canetti, and Hugo Krawczyk. "Keying hash functions for message authentication." Annual international cryptology conference. Springer, Berlin, Heidelberg, 1996.