Towards Formal Verification of HotStuff-Based Byzantine Fault Tolerant Consensus in Agda

NASA Formal Methods 2022

Mark Moir
Architect
Oracle Labs

Joint work with: Harold Carr, Christa Jenkins, Victor Cacciari Miraldo and Lisandra Silva

Agenda

1. Problem and contributions
2. Abstract model and definitions
3. Key theorem, relating it to an implementation
4. Remarks about approach
5. Concluding remarks

Agenda

1. **Problem and contributions**
2. Abstract model and definitions
3. Key theorem, relating it to an implementation
4. Remarks about approach
5. Concluding remarks

Byzantine Fault Tolerant Consensus

- **Consensus: distributed peers (repeatedly) agree on proposed values**
- **Fault tolerant: even if some are “faulty” (e.g., crash)**
- **Byzantine: even if some peers actively and maliciously misbehave**
- **Many proposed BFT consensus solutions in literature, with various properties**
  - Notoriously difficult to get right
  - Many examples of incorrect “solutions”
  - None with fully formal, machine checked proofs
- **New solutions emerging and being adopted**
  - HotStuff (Yin et al., PODC 2019)
  - LibraBFT / DiemBFT (based on HotStuff)
- **Context: we have developed a Haskell implementation based on LibraBFT, and we are working towards formally verifying its correctness**

Contributions

- **Defined abstract model of core protocol underlying HotStuff/LibraBFT**
- **Precisely formulated assumptions**
  - Limits on combined power of dishonest peers
  - Rules that honest peers obey
- **Precisely stated correctness (safety) properties (liveness would be proved for specific implementations, not the abstract model)**
  - Informally, “honest peers agree”
- **Formal, machine-checked proofs**
- **Development is in Agda**
- **Available in open source**
  - https://github.com/oracle/bft-consensus-agda/releases/tag/nasafm2022

Power of abstraction

- **Abstract model knows nothing of message formats, validation, implementation data structures and logic, etc.**
- **Focusing on core protocol enables verifying a range of implementations, without repeating hard work of verifying underlying protocol**
- **LibraBFT under development during verification effort, no need to repeat abstract work when updating our implementation**

Agenda

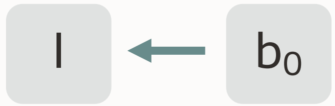
1. Problem and contributions
2. **Abstract model and definitions**
3. Key theorem, relating it to an implementation
4. Remarks about approach
5. Concluding remarks

Growing a chain of Records

- **Start with initial (genesis) record I**

Growing a chain of Records

- **Start with initial (genesis) record I**
- **Extend with alternating Block proposals and Quorum Certificates (QCs) to “certify” them**

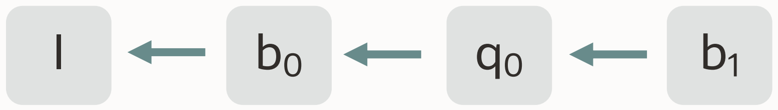


Growing a chain of Records

- **Start with initial (genesis) record I**
- **Extend with alternating Block proposals and Quorum Certificates (QCs) to “certify” them**

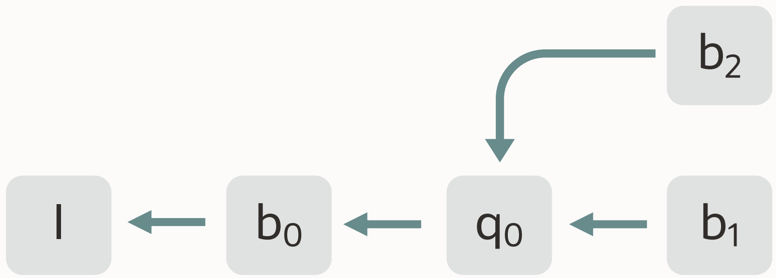
Growing a chain of Records

- **Start with initial (genesis) record I**
- **Extend with alternating Block proposals and Quorum Certificates (QCs) to “certify” them**



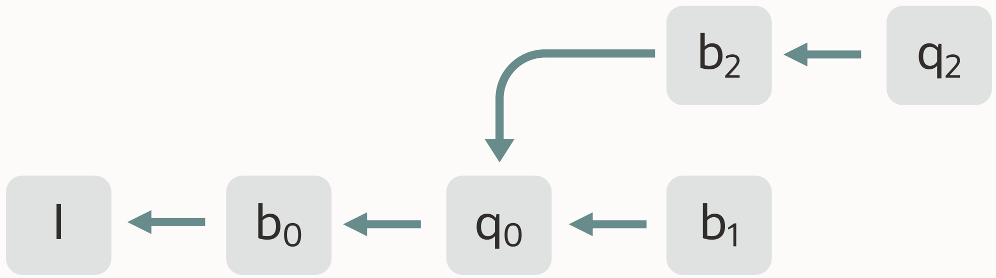
Growing a chain of Records

- Start with initial (genesis) record I
- Extend with alternating Block proposals and Quorum Certificates (QCs) to "certify" them
- If QC not formed/known, time out and propose alternative Block



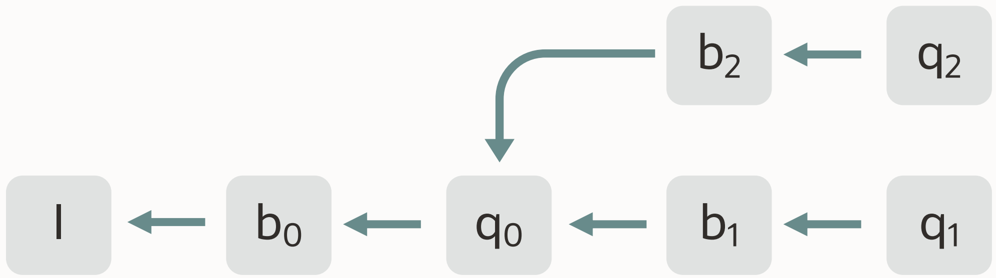
Growing a chain of Records

- **Start with initial (genesis) record I**
- **Extend with alternating Block proposals and Quorum Certificates (QCs) to “certify” them**
- **If QC not formed/known, time out and propose alternative Block**



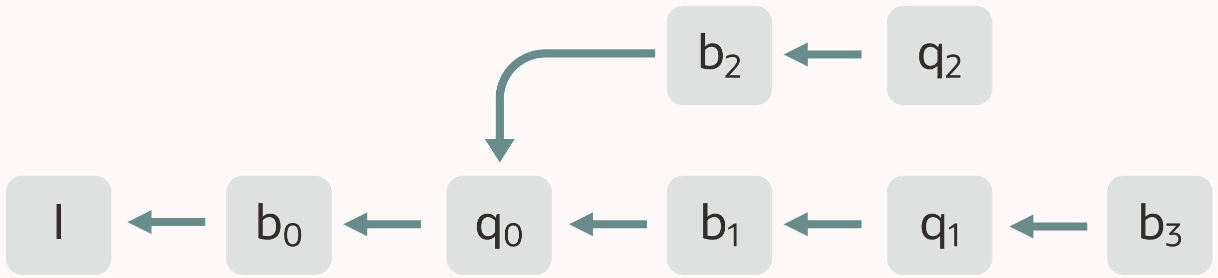
Growing a chain of Records

- **Start with initial (genesis) record I**
- **Extend with alternating Block proposals and Quorum Certificates (QCs) to "certify" them**
- **If QC not formed/known, time out and propose alternative Block**
- **What if QC does emerge for b_1?**



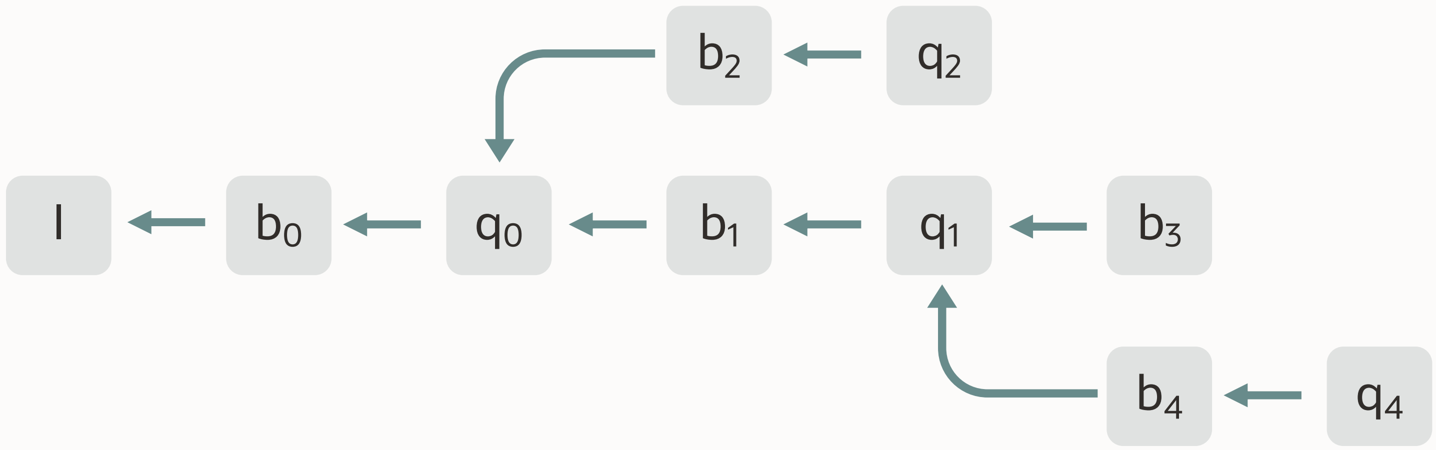
Growing a chain of Records

- Start with initial (genesis) record I
- Extend with alternating Block proposals and Quorum Certificates (QCs) to “certify” them
- If QC not formed/known, time out and propose alternative Block
- What if QC does emerge for b_1?



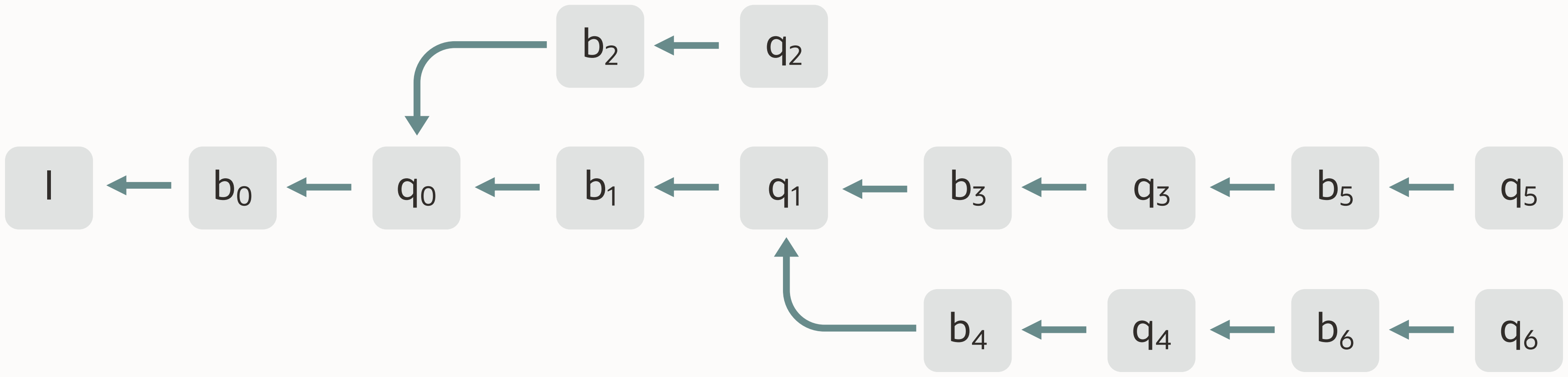
Growing a chain of Records

- Start with initial (genesis) record I
- Extend with alternating Block proposals and Quorum Certificates (QCs) to “certify” them
- If QC not formed/known, time out and propose alternative Block
- What if QC does emerge for b_1?



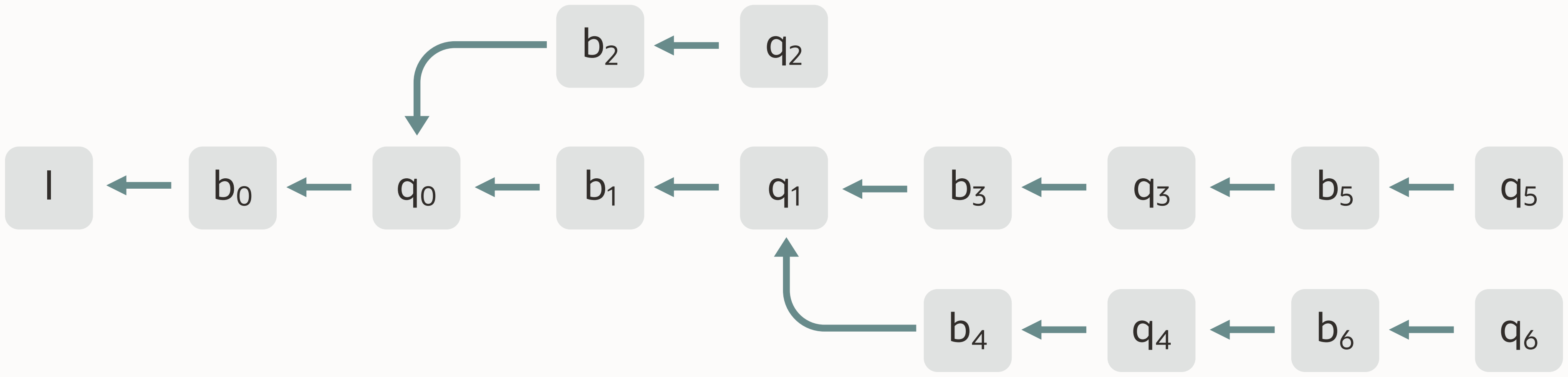
Growing a chain of Records

- Start with initial (genesis) record **I**
- Extend with alternating **Block** proposals and Quorum Certificates (**QC**s) to "certify" them
- If **QC** not formed/known, time out and propose alternative **Block**
- What if **QC** does emerge for b_1?



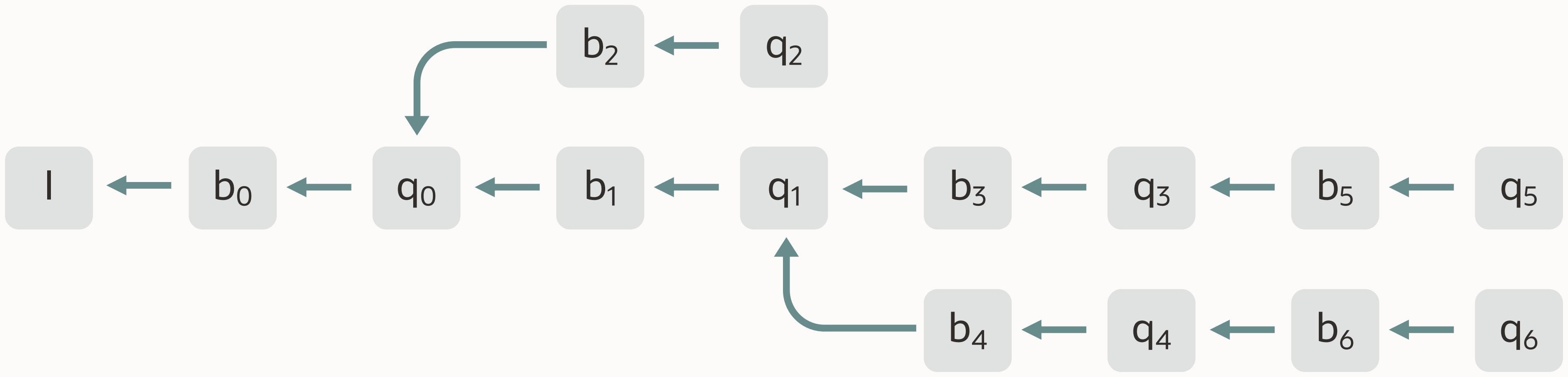
Growing a ~~chain~~ tree of Records

- **Start with initial (genesis) record I**
- **Extend with alternating Block proposals and Quorum Certificates (QCs) to "certify" them**
- **If QC not formed/known, time out and propose alternative Block**
- **What if QC does emerge for b_1?**
- **Must model a <u>tree</u> of Records**



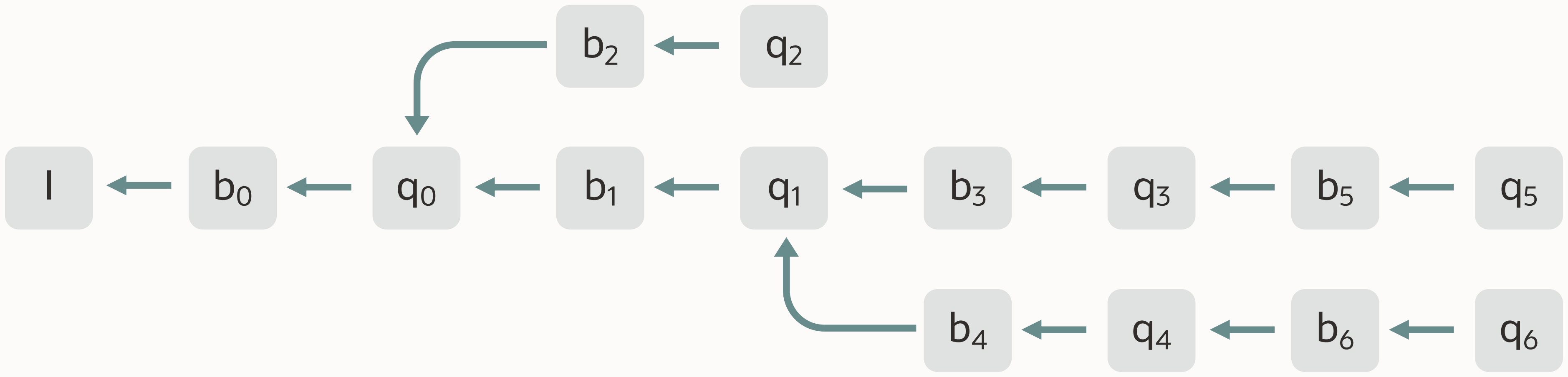
Growing a ~~chain~~ tree of Records

- Start with initial (genesis) record I
- Extend with alternating Block proposals and Quorum Certificates (QCs) to "certify" them
- If QC not formed/known, time out and propose alternative Block
- What if QC does emerge for b_1?
- Must model a <u>tree</u> of Records
- How to ensure honest peers agree on decisions?



Desired property, less informally

- **If two honest peers each commit (decide on) a Block, then the Blocks do not "conflict": there is a single path in the tree that contains them both:**
  - Commit b_1 and b_6: no problem
  - Commit b_2 and b_6: conflict!



Abstract records

- **QCs indicate UID and round of Blocks they certify**
- **To enable specifying rules to ensure consistent decisions Blocks have round numbers**
- **QCs and Votes they contain indicate the Round of the voted-for Block**

```
data Record : Set where
  I  :                Record
  B : Block     → Record
  Q : QC         → Record
```

```
record Block  : Set where
  field
    bRound   : Round
    bId         : UID
    bPrevQC  : Maybe UID
```

```
record QC : Set where
  field
    qRound          : Round
    qCertBlockId  : UID
    qVotes           : List Vote
    …
```

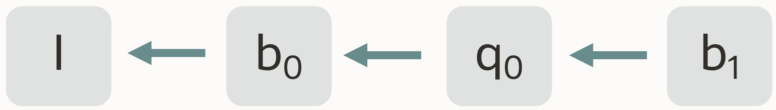
```
record Vote : Set where
  field
    abs-vRound       : Round
    abs-vMember    : …
    abs-vBlockUID  : UID
```

Extends relation (_←_)

- _←_ imposes constraints on rounds
- **QC** is for same round as **Block** it extends
- **Block** is for higher round than **Block** it extends (via a **QC**)

```
data _←_ : Record → Record → Set where
  I←B : {b : Block}
      → 0 < getRound b
      → bPrevQC b ≡ nothing
      → I ← B b
  B←Q : {b : Block} {q : QC}
      → getRound q ≡ getRound b
      → bId b ≡ qCertBlockId q
      → B b ← Q q
  Q←B : {q : QC} {b : Block}
      → getRound q < getRound b
      → just (qCertBlockId q) ≡ bPrevQC b
      → Q q ← B b
```

Defining RecordChains



RecordChain (B b_1)

```
data RecordChainFrom (o : Record) :
          Record → Set where
  empty : RecordChainFrom o o
  step   : ∀ {r r'}
           → (rc : RecordChainFrom o r)
           → r ← r'
           → RecordChainFrom o r'

RecordChain : Record → Set
RecordChain = RecordChainFrom I
```

Defining 𝕂-Chains

```
data 𝕂-chain (R : ℕ → Record → Record → Set)
    : (k : ℕ){o r : Record} → RecordChainFrom o r → Set where
  0-chain : ∀{o r}   {rc : RecordChainFrom o r} → 𝕂-chain R 0 rc
  s-chain  : ∀{k o r}{rc : RecordChainFrom o r}{b : Block}{q : QC}
         → (r←b  : r   ← B b)
         → (prf   : R k r (B b))
         → (b←q : B b ← Q q)
         → 𝕂-chain R k rc
         → 𝕂-chain R (suc k) (step (step rc r←b) b←q)

-- Contiguous 𝕂-chains are those in which all adjacent pairs of
-- Records have contiguous rounds.
 Contig : ℕ → Record → Record → Set
 Contig 0          _ _ = Unit
 Contig (suc _) r  r' = round r' ≡ suc (round r)
```

Roughly speaking,

𝕂-Chain Contig k rc

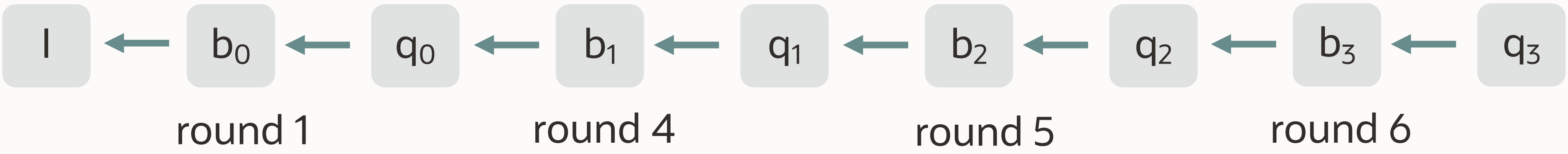
Says that rc contains at least k Blocks, such that the Rounds of the last k Blocks are consecutive

How to decide a Block is committed?

```
data CommitRuleFrom {o r : Record}(rc : RecordChainFrom o r)(b : Block) : Set where
  commit-rule : (c3 : 𝕂-chain Contig 3 rc)
                → b ≡ c3 b⟦ suc (suc zero) ⟧
                → CommitRuleFrom rc b

CommitRule : ∀{r} → RecordChain r → Block → Set
CommitRule = CommitRuleFrom
```

rc : RecordChain (Q q_3)



CommitRule rc b_1

Block b_1 is now committed (and all Blocks before it, i.e., b_0)

Agenda

1. Problem and contributions
2. Abstract model and definitions
3. **Key theorem, relating it to an implementation**
4. Remarks about approach
5. Concluding remarks

Key theorem: thmS5

- **If we have CommitRules for Blocks b and b' enabling commiting both Blocks, then the one of the Blocks is in the RecordChain of the other's CommitRule (i.e., there is no conflict in committing them both)…**

```
thmS5 : ∀ {q q'}
      → {rc  : RecordChain (Q q )} → All-InSys rc
      → {rc' : RecordChain (Q q')} → All-InSys rc'
      → {b b' : Block}
      → CommitRule rc  b
      → CommitRule rc' b'
      → NonInjective-≡-pred (InSys ∘ B) bId
      ⊎ (B b) ∈RC rc'
      ⊎ (B b') ∈RC rc
```

- **…unless there are two different Blocks "in the system" with the same block ID.**

Relating thmS5 to an implementation

- **To invoke thmS5 for a particular implementation, we must instantiate these module parameters:**

```
module LibraBFT.Abstract.RecordChain.Properties
  (UID    : Set)                                      -- type for Block ids
  …
  …
  (𝓔                    : EpochConfig UID …)          -- specifies peers, assumptions, …
  …
  (InSys                : Record → Set …)             -- which abstract Records are represented
                                                          (e.g., in messages that have been sent)
  (votes-only-once      : VotesOnlyOnceRule InSys)    -- honest peers obey two rules
  (preferred-round-rule : PreferredRoundRule InSys)
where
  …
  thmS5 : …
  … proof of thmS5
```

Rules for honest peers (1/2)

- **An honest peer does not send inconsistent Votes for the same Round:**

```
VotesOnlyOnceRule : Set …
VotesOnlyOnceRule
  = (α : Member) → Meta-Honest-Member α
    → ∀{q q'} → InSys (Q q) → InSys (Q q’)
    → (v  : α ∈QC q) (v' : α ∈QC q')
    → abs-vRound (∈QC-Vote q v) ≡ abs-vRound (∈QC-Vote q’ v’)
    → ∈QC-Vote q v ≡ ∈QC-Vote q' v'
```

- **Manual proof in an early LibraBFT paper required “Increasing Round” constraint:**
  An honest node that voted once for B in the past may only vote for B’ if round (B) < round (B’)
- **One contribution is making rules *precise* enough to enable rigorous (machine-checked) proofs**

Rules for honest peers (2/2)

```
PreferredRoundRule : Set …
PreferredRoundRule
    = ∀(α : Member) → Meta-Honest-Member α
    → ∀{q q'}
    → {rc : RecordChain (Q q)} → All-InSys rc
    → {n : ℕ} (c3 : 𝕂-chain Contig (3 + n) rc)
    → (v : α ∈QC q)
    → {rc' : RecordChain (Q q')} → All-InSys rc'
    → (v' : α ∈QC q')
    → abs-vRound (∈QC-Vote q v) < abs-vRound (∈QC-Vote q' v')
    → NonInjective-≡-pred (InSys ∘ B) bId
    ⊎ (getRound (kchainBlock (suc (suc zero)) c3) ≤ prevRound rc')
```

- **The key rule that honest peers must follow to avoid contributing to QCs that could result in committing conflicting Blocks**
- **Key implementation requirement for invoking thmS5**
- **Again, result required only if there is no injectivity failure among Blocks "in the system"**

Agenda

1. Problem and contributions
2. Abstract model and definitions
3. Key theorem, relating it to an implementation
4. **Remarks about approach**
5. Concluding remarks

Block id injectivity

- Abstract model and proofs know nothing about UID, how Block ids are assigned, etc.
- Typical implementations use cryptographic hash functions (e.g., SHA256)
- Standard to assume that a computationally bounded adversary cannot find two different values (e.g., Blocks) that hash to the same value
- Most related work implicitly or explicitly assumes that such hash collisions do not exist, which is false by an easy counting argument (hash results are of fixed size)
- False implies anything. Danger!
- We don't assume hash functions are injective. Instead, we ensure that our results hold unless and until there is a collision between Blocks that the implementation considers "in the system" (e.g., messages containing them have been sent).
- ToyChain (Pîrlea and Sergey, CPP 2018) work to address this issue required changes to every proof, one ballooned from 10 lines to 150!

Power of abstraction

- **Abstract model also knows nothing of message formats, validation, implementation data structures and logic, etc.**
- **Focusing on core protocol enables verifying a range of implementations, without repeating hard work of verifying underlying protocol**
- **LibraBFT under development during verification effort, no need to repeat abstract work when updating our implementation**

Power of abstraction

- **Abstract model also knows nothing of message formats, validation, implementation data structures and logic, etc.**
- **Focusing on core protocol enables verifying a range of implementations, without repeating hard work of verifying underlying protocol**
- **LibraBFT under development during verification effort, no need to repeat abstract work when updating our implementation**
- **That said, recently LibraBFT changed to a** CommitRule **based on 2-chains. That implementation does not ensure the** PreferredRoundRule **defined in our development**
- **Does not mean it is not correct, but that it cannot be proved correct using our abstract model**
- **Updating our work for a 2-chain-based** CommitRule **is future work**

Agenda

1. Problem and contributions
2. Abstract model and definitions
3. Key theorem, relating it to an implementation
4. Remarks about approach
5. **Concluding remarks**

Concluding remarks

- **Byzantine Fault Tolerant consensus is notoriously difficult to get right**
- **Formal, machine-checked verification is important, but difficult and time consuming**
- **Our approach separates correctness of underlying protocol from details of a range of (but not all) implementations**
- **We avoid the dangerous assumption that hash collisions do not exist (they do!) by tying hash collisions to specific values actually encountered in an execution**
- **Results for a single "epoch", epoch change / reconfiguration is future work**
- **Broader project includes:**
  - Agda translation of our Haskell implementation
  - Syntax and library support to keep Agda close to Haskell implementation
  - System Model and machinery for modeling and proving properties about a distributed system in which honest peers follow implementation, dishonest ones unconstrained other than inability to forge honest signatures
  - Significant progress towards verifying that our implementation satisfies requirements to instantiate abstract results
  - Open source: https://github.com/oracle/bft-consensus-agda/releases/tag/nasafm2022

Questions?

Mark Moir (`mark.moir@oracle.com`)

Architect
Oracle Labs