CS 591: Formal Methods in Security and Privacy

Proofs of Protocol Security in Real/Ideal Paradigm

Marco Gaboardi
gaboardi@bu.edu

Alley Stoughton
stough@bu.edu

Real/Ideal Paradigm

- We proved IND-CPA (indistinguishability under chosen plaintext attack) security of a symmetric encryption scheme built from a pseudorandom function plus randomness
- Now, we're going to consider a proof in the Real/Ideal Paradigm of the security of a three party cryptographic protocol
- In the real/ideal paradigm there are two games:
  - A "real" game based on how the actual protocol works
  - An "ideal" game that is secure by construction
- In the security proof we show that an Adversary can't distinguish the two games—or can only distinguish them with negligible probability

Private Count Retrieval Protocol

- The Private Count Retrieval (PCR) Protocol involves three parties:
  - a Server, which holds a database
  - a Client, which makes queries about the database
  - an *untrusted* Third Party (TP), which mediates between the Server and Client
- A database is one-dimensional: it consists of a list of elements
- Each query is also an element, and is a request for the count of the number of times it occurs in the database

Private Count Retrieval Protocol

- For example, suppose the database is `[0; 2; 0; 4; 2]`.
- If the query is `0`, the answer is:
  - `2`
- If the query is `4`, the answer is: `1`
- If the query is `3`, the answer is:
  - `0`

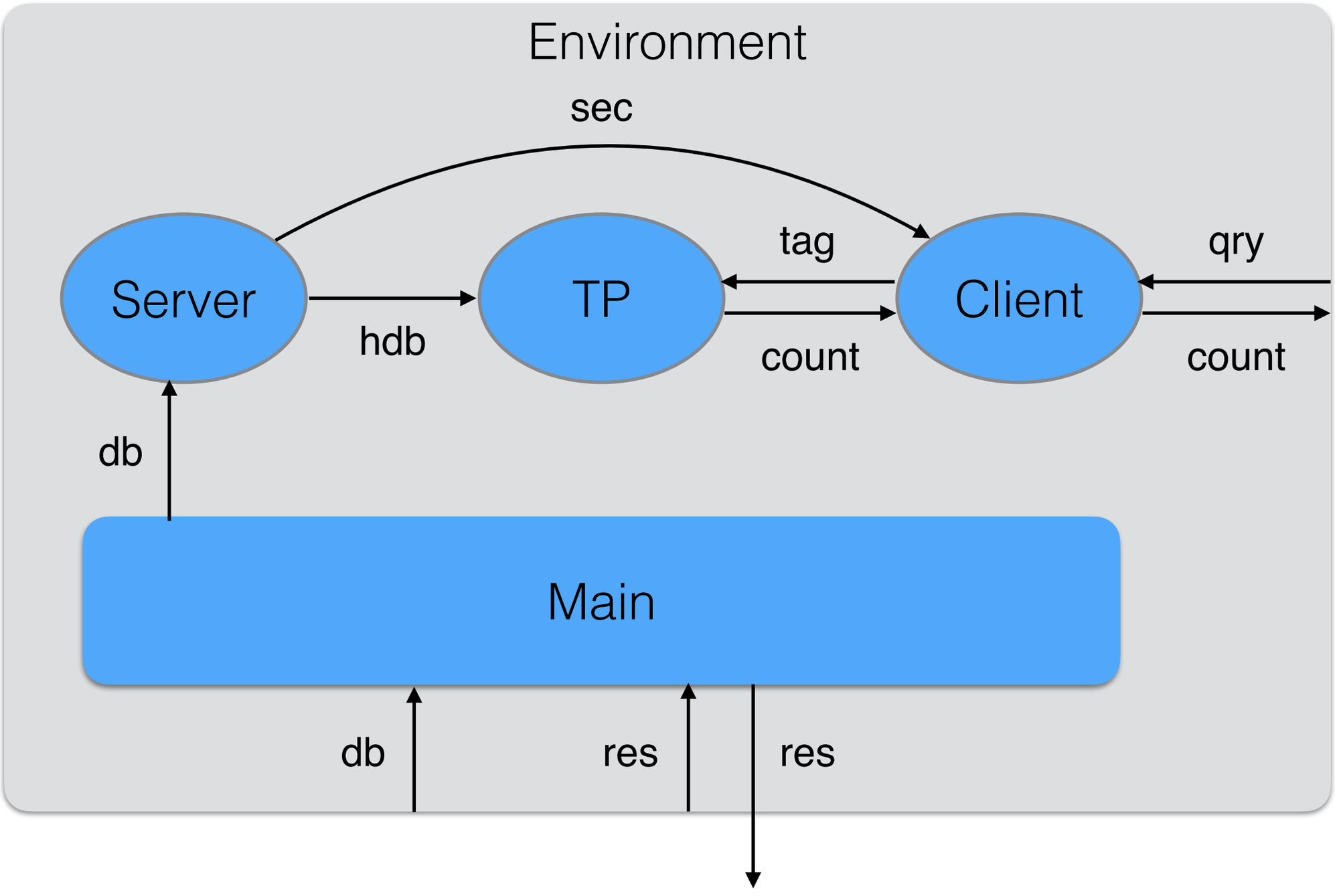
Security Goals for PCR

- Informally, the goal is for:
  - Client to only learn the counts for its queries, not anything else about the database (we'll limit how many queries it can make)
  - Server to learn nothing about the queries made by the Client other than the number of queries that were made
  - TP to learn nothing about the database and queries other than certain element *patterns*

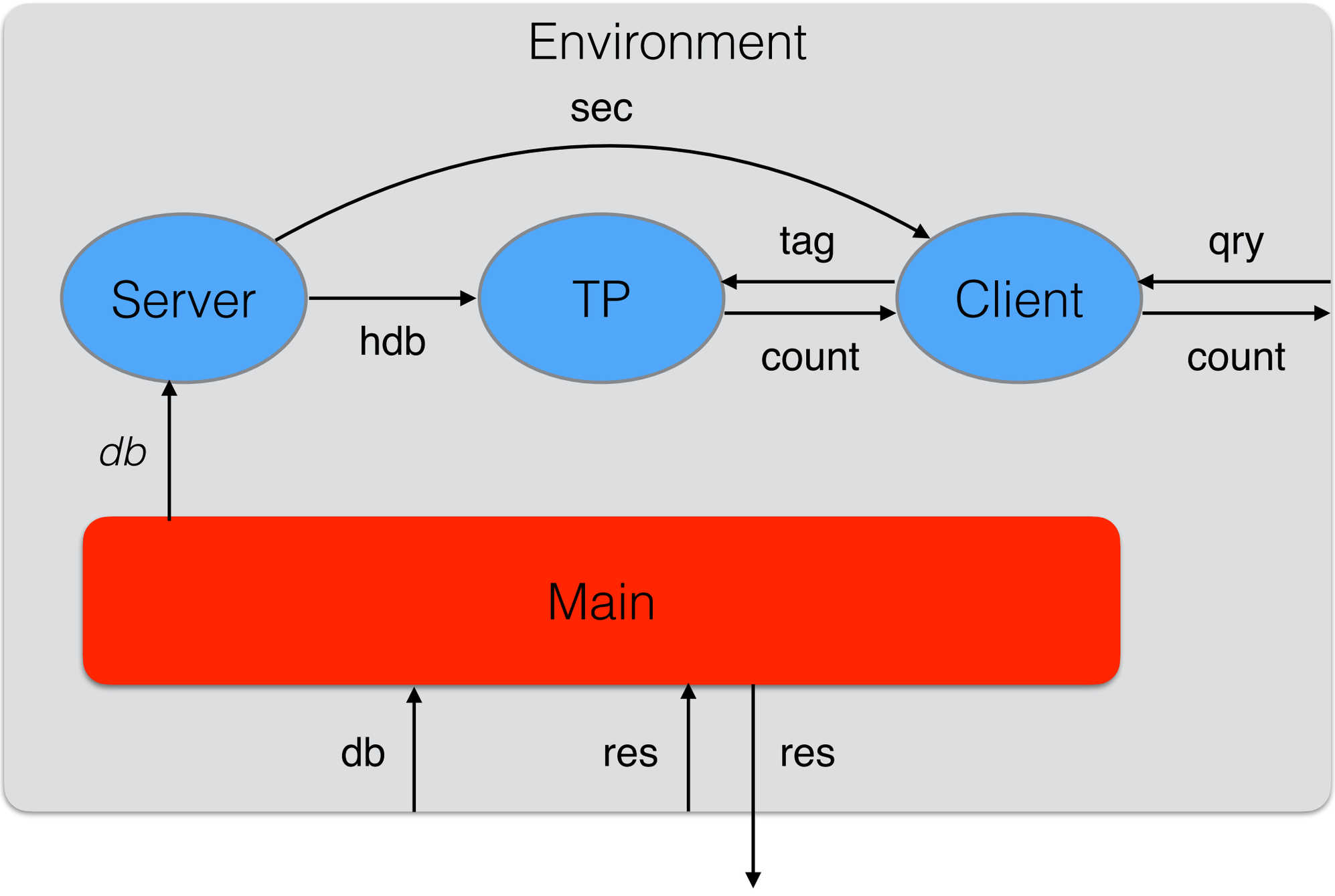
Hashing

- The PCR protocol makes use of *hashing*, a process transforming a value of some type into a bit string of a fixed length
  - When distinct inputs are hashed, it should be very unlikely that the resulting bit strings are equal
  - Given a bit string, it should be hard to find an input that hashes to it
- In an implementation, we might use a member of the SHA family of hash functions
- But in our proofs, we'll model hashing via a *random oracle*

PCR Protocol Operation

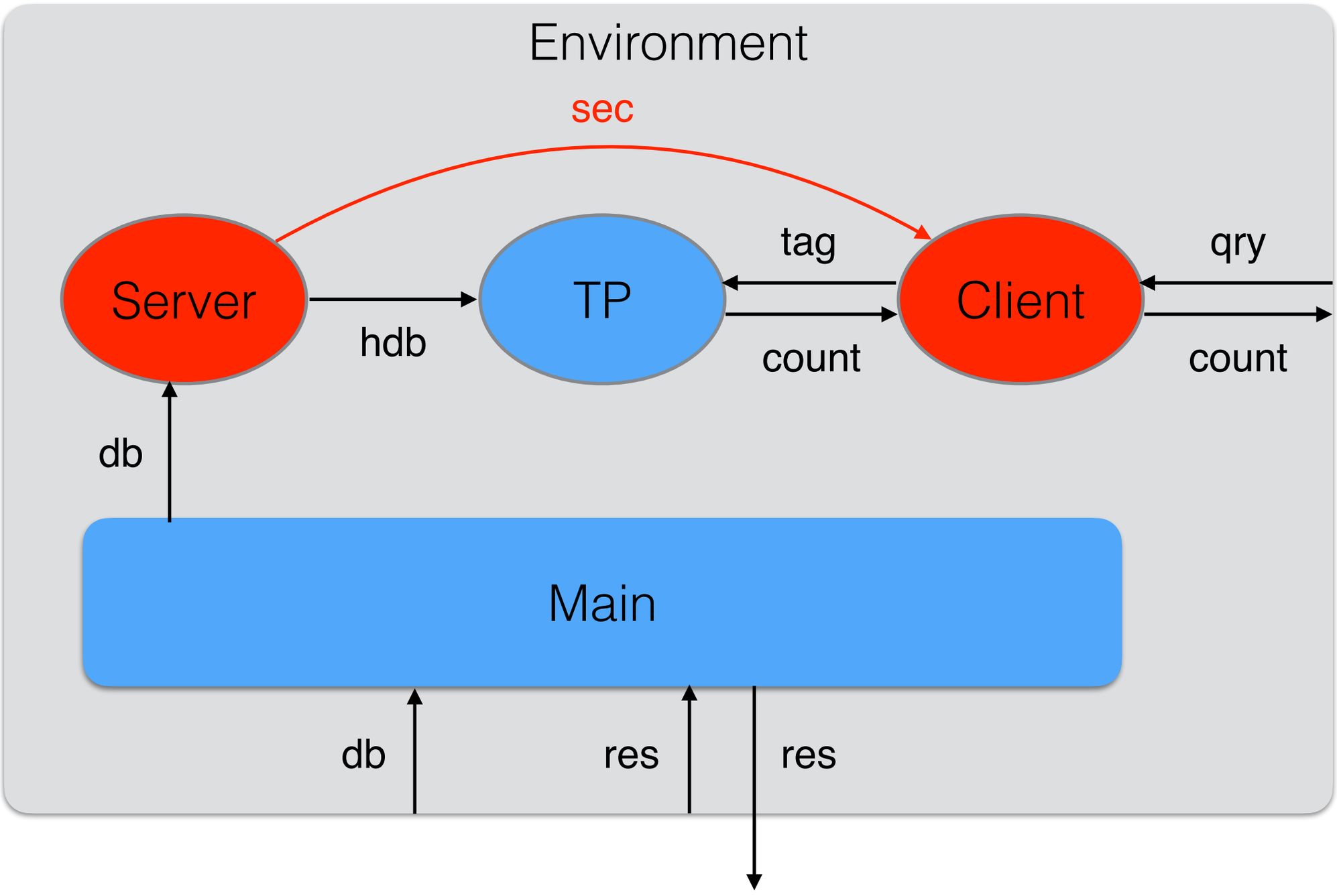


PCR Protocol Operation

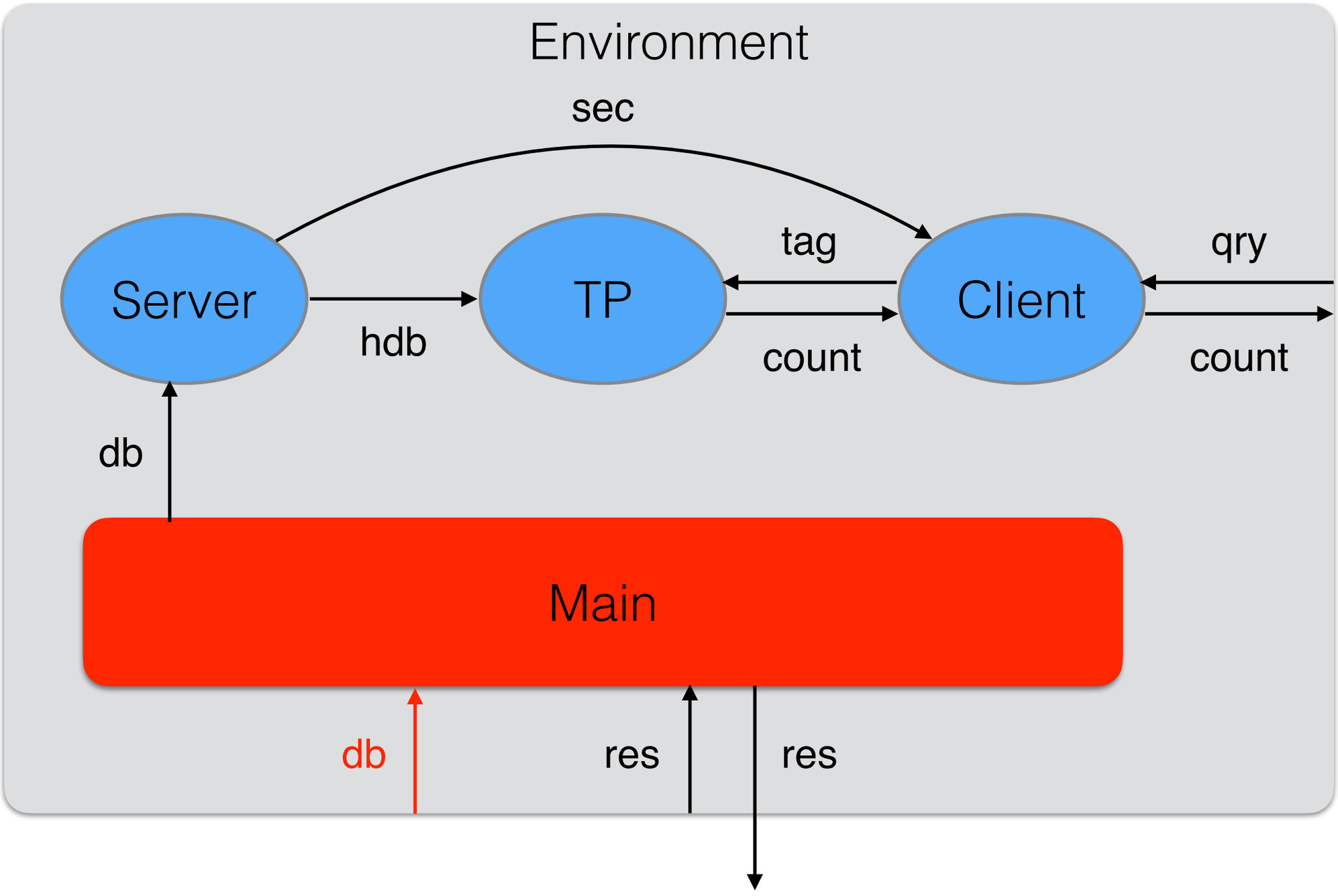


PCR Protocol Operation

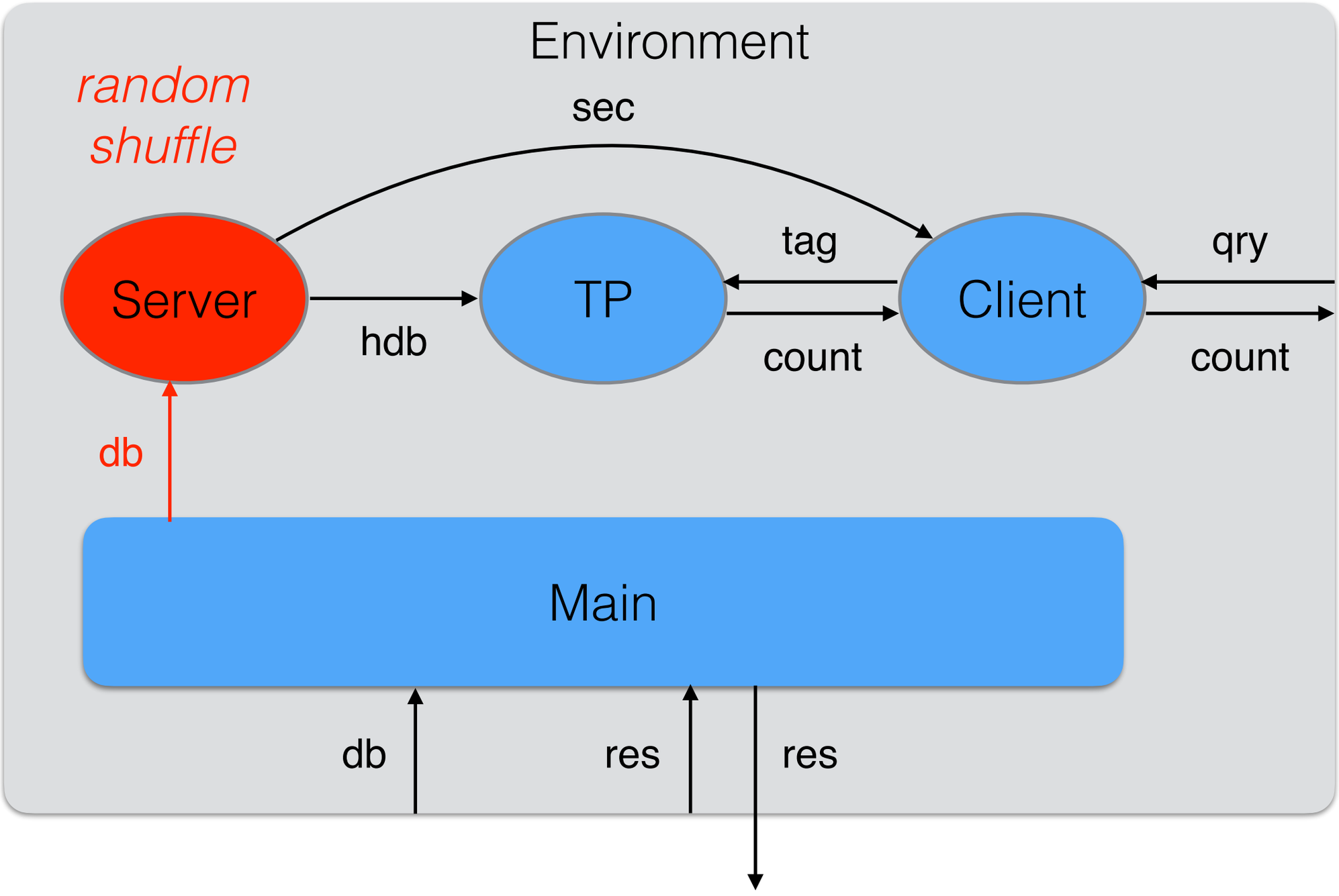
secrets are bit strings of length **sec_len**



PCR Protocol Operation

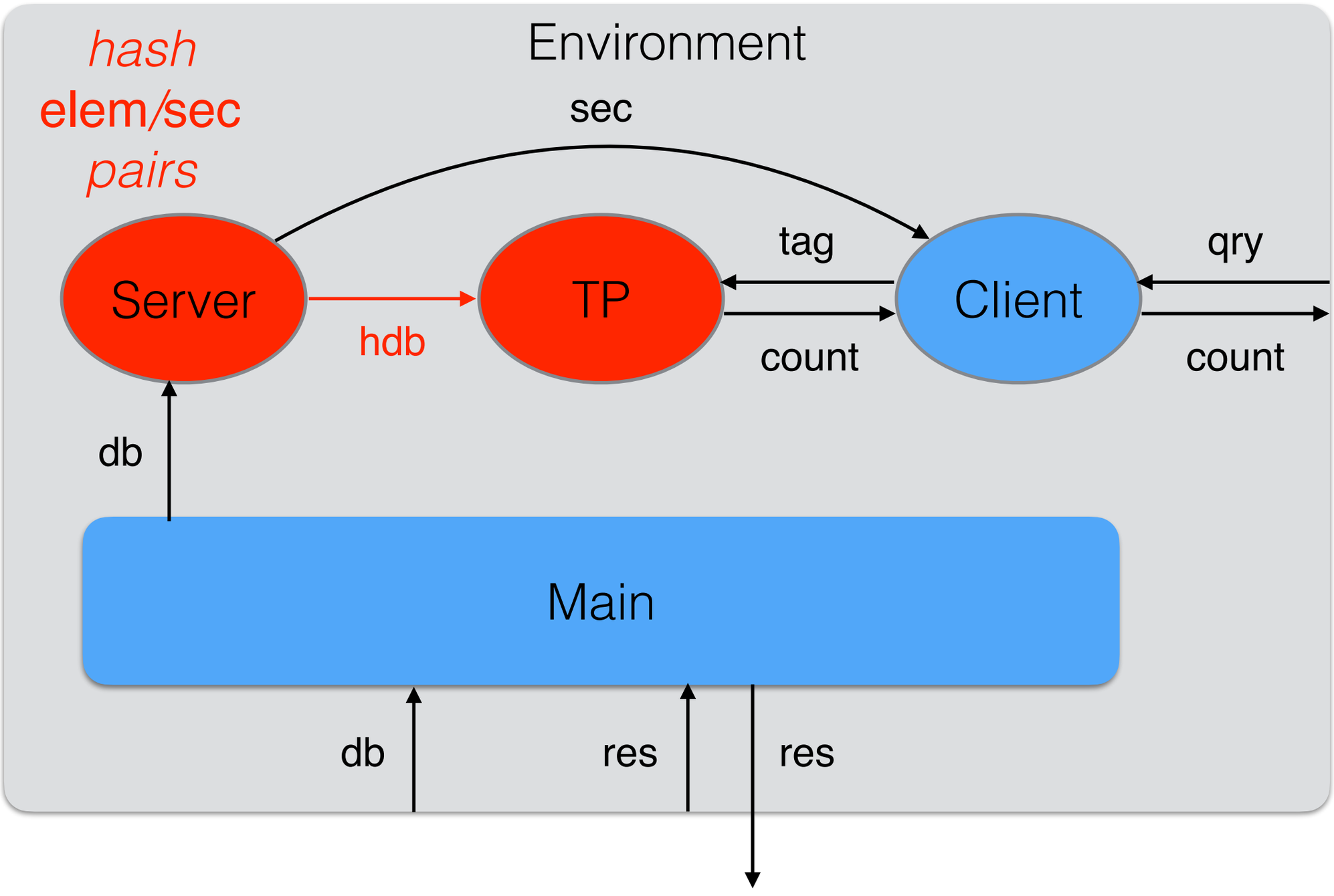


PCR Protocol Operation

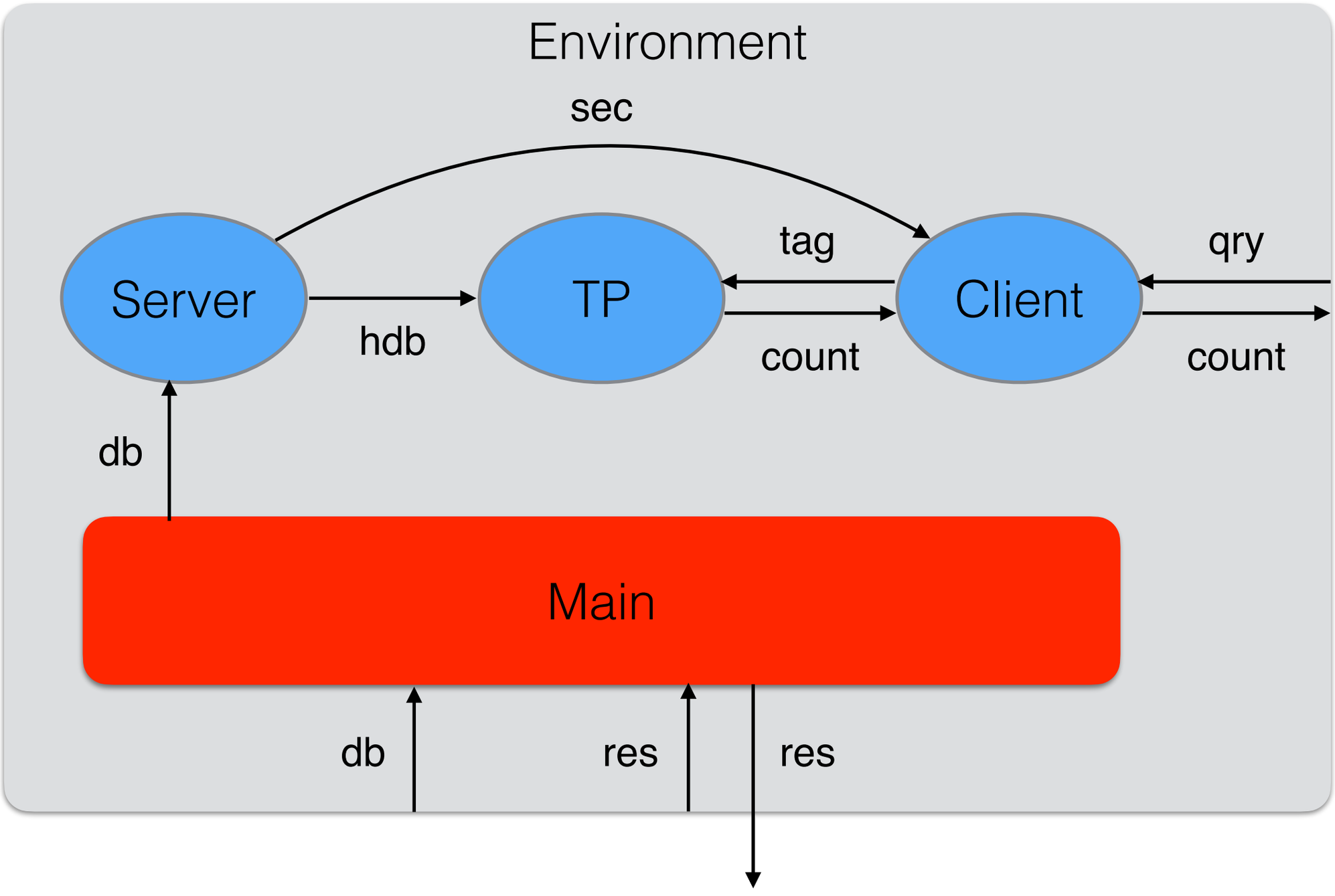


PCR Protocol Operation

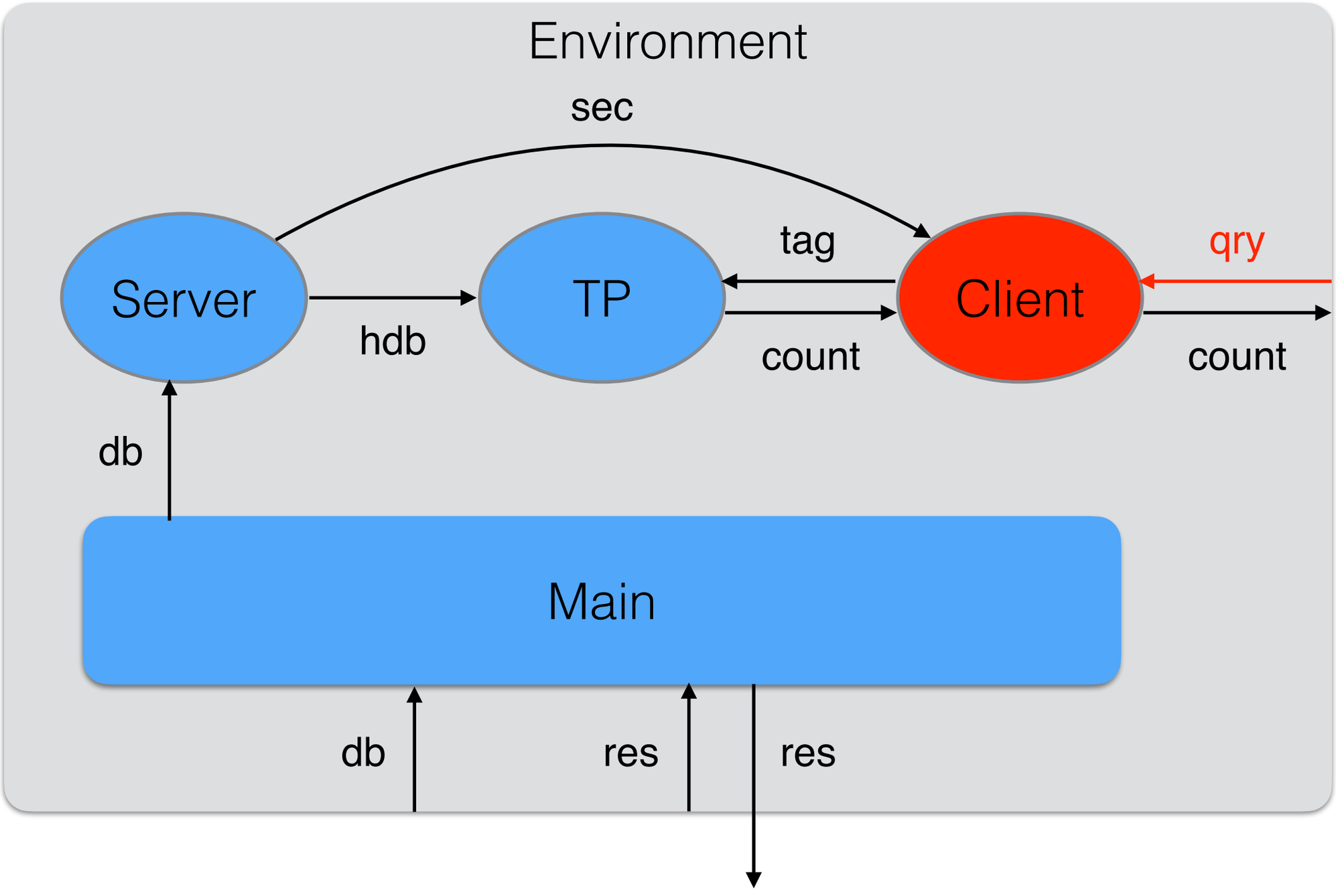
tags are bit strings of length **tag_len**



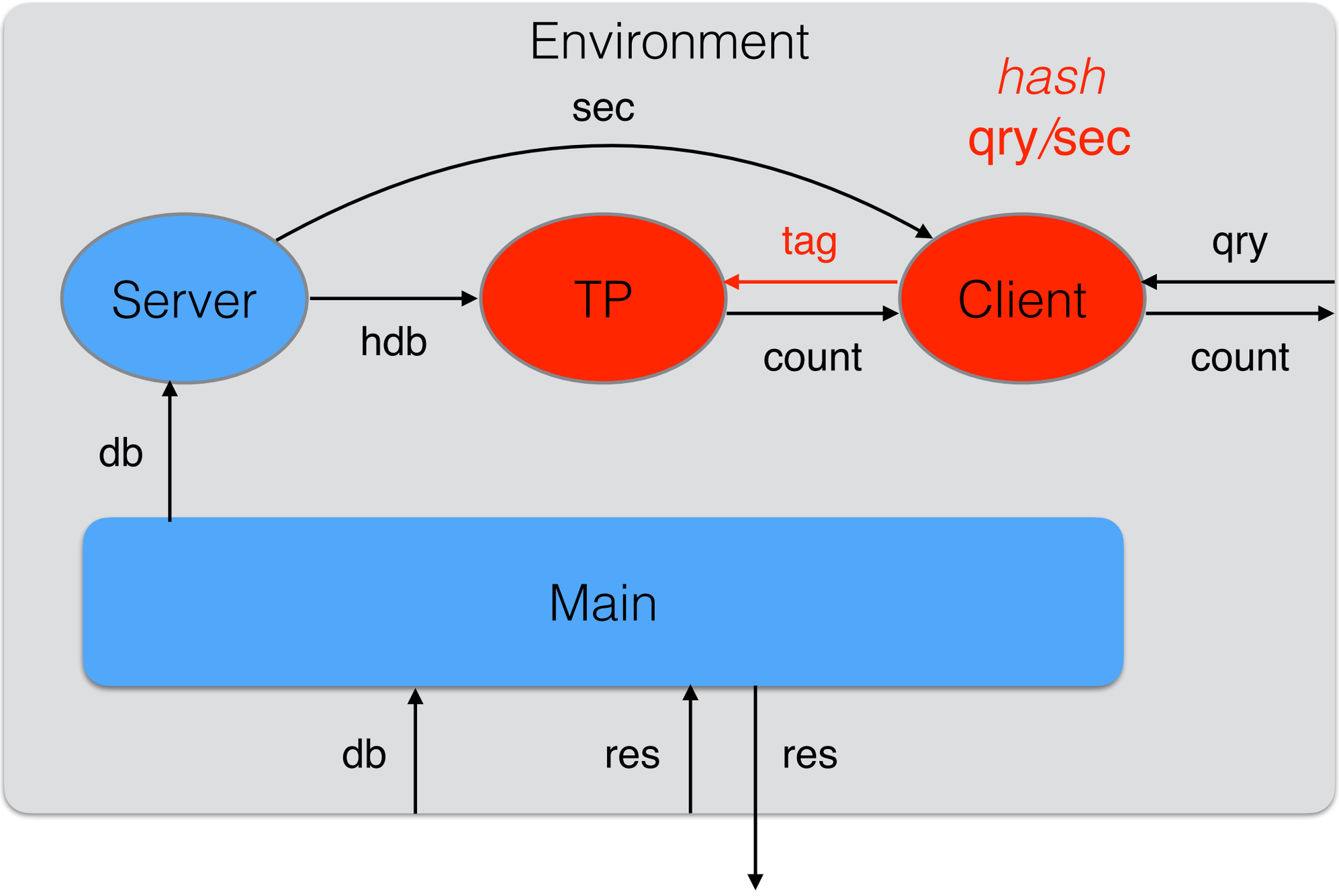
PCR Protocol Operation



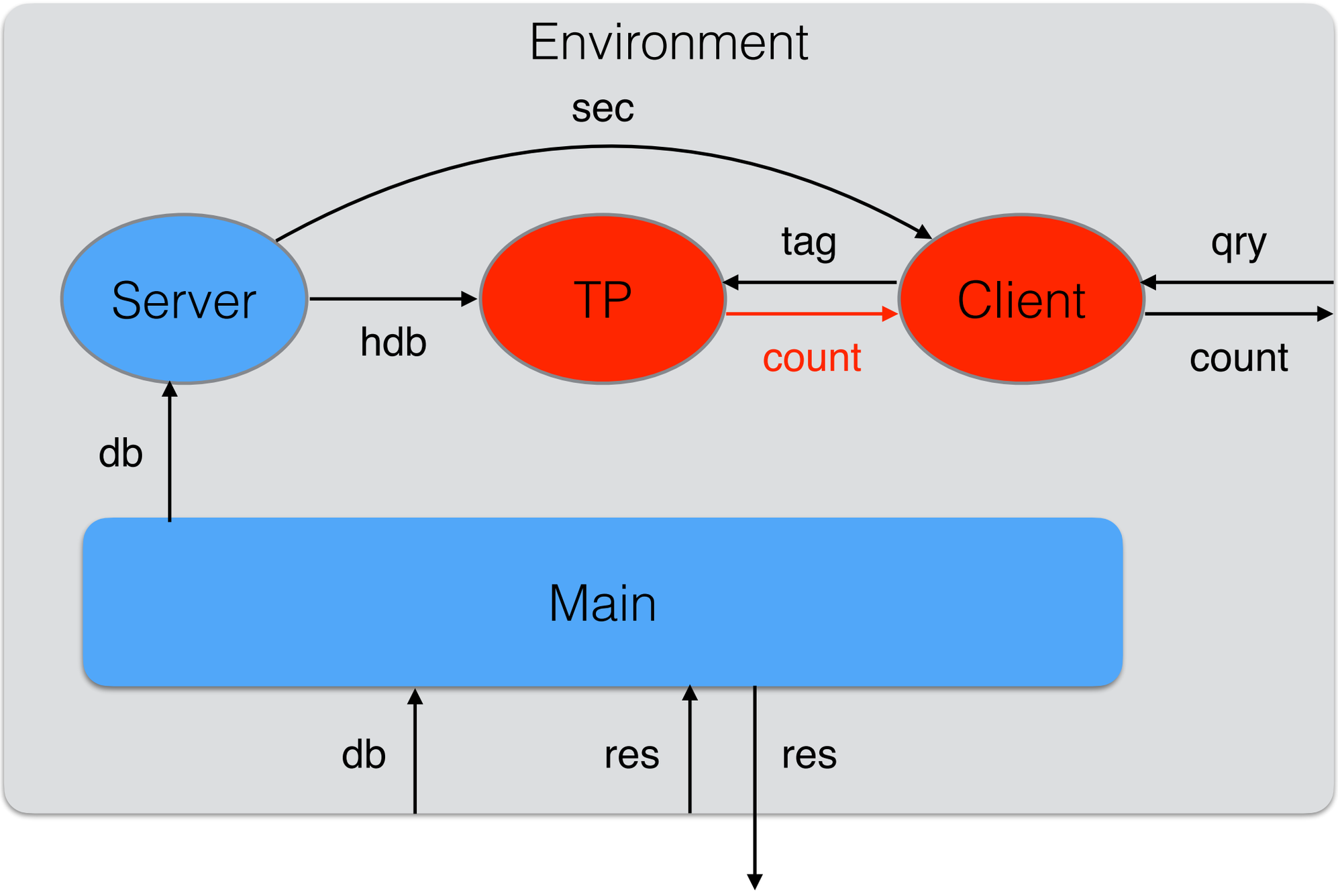
PCR Protocol Operation



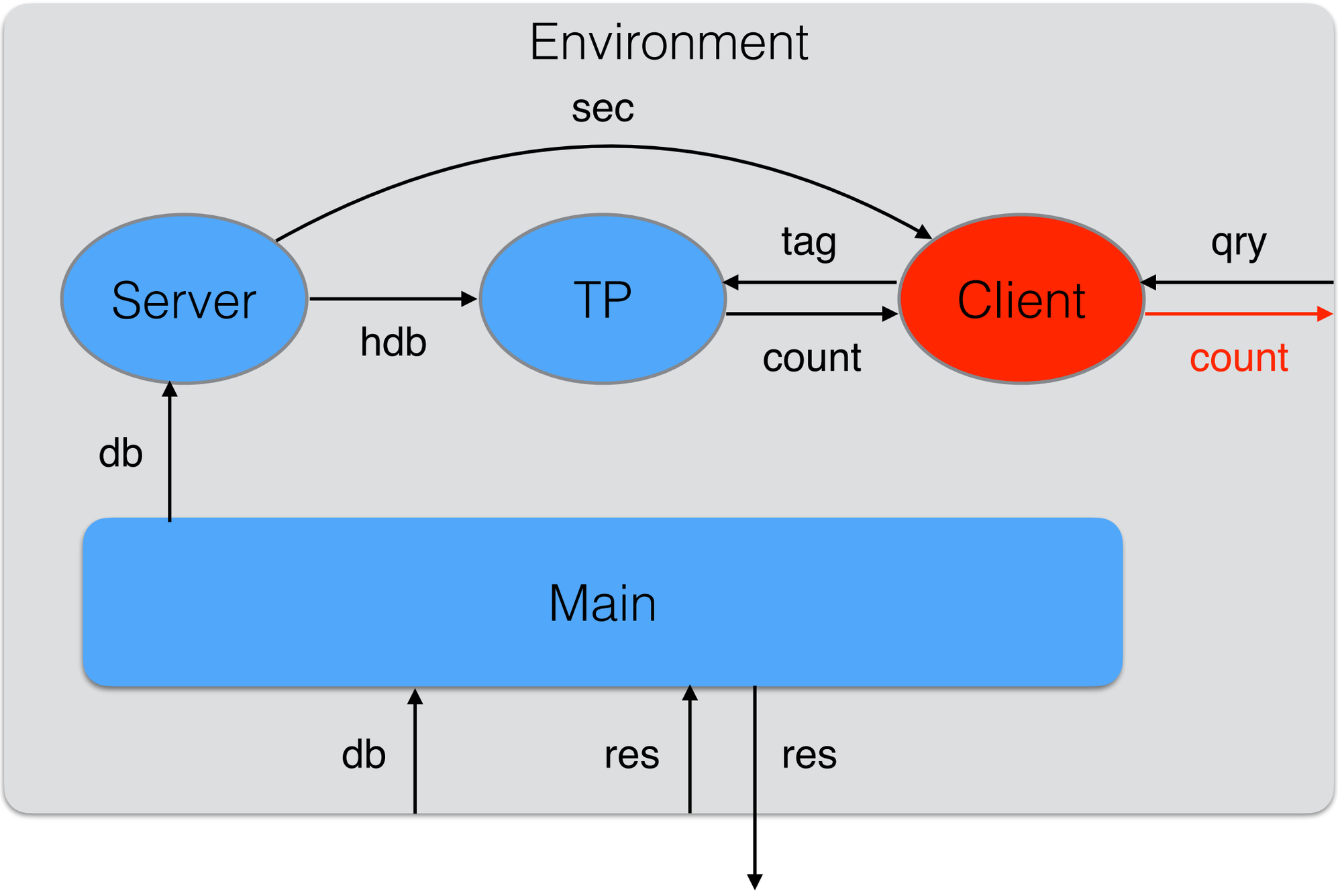
PCR Protocol Operation



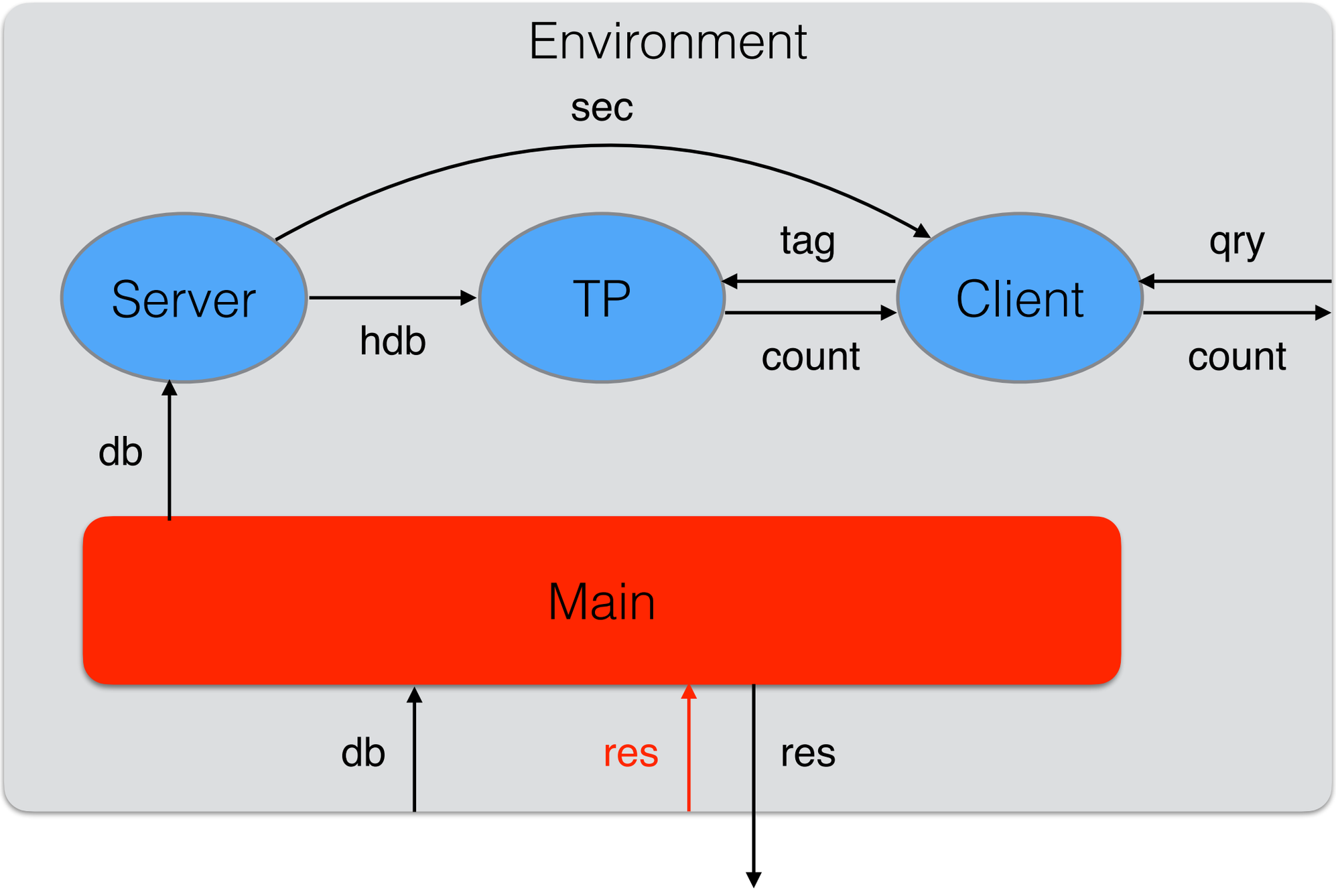
PCR Protocol Operation



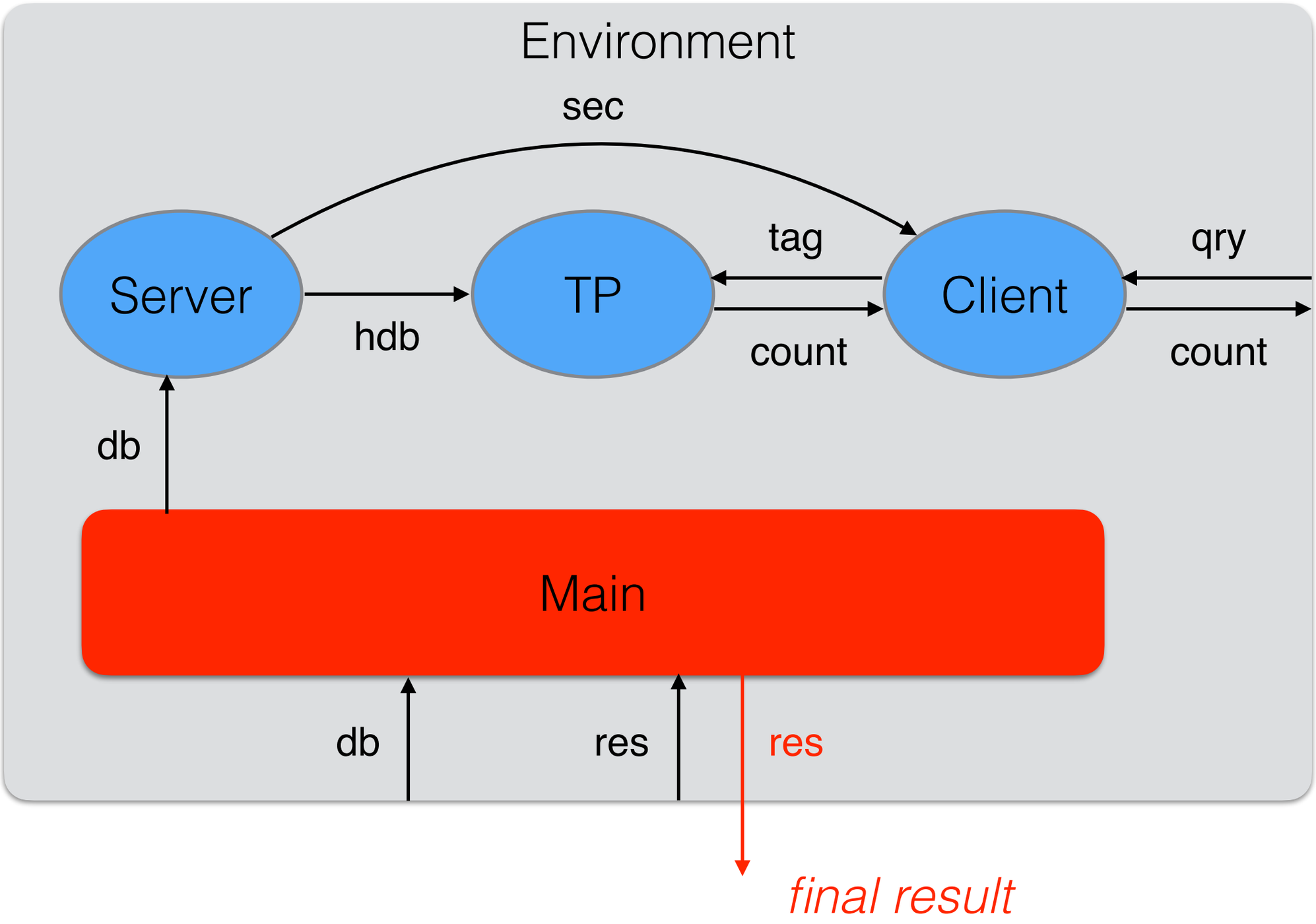
PCR Protocol Operation



PCR Protocol Operation



PCR Protocol Operation



Protocol Example

- E.g., suppose the original database was `[0; 1; 1; 2]` and the queries are `1`, `2` and `3`
- The Server's shuffled database might be `[1; 0; 2; 1]`
- TP will get a hashed database $[t_2; t_1; t_3; t_2]$ and hash tags t_2, t_3 and t_4, and so will return to Client counts `2`, `1` and `0` (assuming no hash collisions)

EasyCrypt Code

- On GitHub you can find:
  - All the EasyCrypt definitions and proofs
  - A link to a conference paper about PCR and its proofs
    - Joint work with Mayank Varia

https://github.com/alleystoughton/PCR

Elements, Secrets and Hashing in EasyCrypt

- Elements (type `elem`) may be anything
- Secrets (type `sec`) are bits strings of length `sec_len`
- Hash tags (type `tag`) are bit strings of length `tag_len`
- Hashing is done using a random oracle in which element/ secret pairs are hashed to hash tags

Random Oracle Theory `RandomOracle`

```
module type OR = {
  proc init() : unit
  proc hash(inp : input) : output
}.

module Or : OR = {
  var mp : (input, output) fmap

  proc init() : unit = {
    mp <- empty;
  }

  proc hash(inp : input) : output = {
    if (! dom mp inp) {
      mp.[inp] <$ output_distr;
    }
    return oget mp.[inp];
  }
}.
```

Random Oracle

```
clone RandomOracle as RO with
  type input <- elem * sec,
  op input_default <- (elem_default, zeros_sec),
  op output_len <- tag_len,
  type output <- tag,
  op output_default <- zeros_tag,
  op output_distr <- tag_distr
proof *.
(* realization *) … (* end *)
```

Thus **`RO.Or`** with module type **`RO.OR`** is the random oracle

Random Shuffling

```
module Shuffle = {
  proc shuffle(xs : elem list) : elem list = {
    var ys : elem list; var i : int;
    ys <- [];
    while (0 < size xs) {
      i <$ [0 .. size xs - 1];
      ys <- ys ++ [nth elem_default xs i];
      xs <- trim xs i;
    }
    return ys;
  }
}.
```

each of the (`size xs`)! reorderings of `xs` are equally possible (because of duplicates, some of these reorderings may be the same)

PCR Protocol

```
type db = elem list. type hdb = tag list.

…
type server_view = server_view_elem list.
type tp_view = tp_view_elem list.
type client_view = client_view_elem list.

module type ENV = {
  proc * init_and_get_db() : db option
  proc get_qry() : elem option
  proc put_qry_count(cnt : int) : unit
  proc final() : bool
}.
```

Each party has a *view* variable that records everything it sees

PCR Protocol

```
module Protocol (Env : ENV) = {
  module Or = RO.Or
  …
  proc main() : bool = {
    var db_opt : db option; var b : bool;
    init_views(); Or.init();
    server_gen_sec(); client_get_sec();
    db_opt <@ Env.init_and_get_db();
    if (db_opt <> None) {
      server_hash_db(oget db_opt);
      tp_get_hdb();
      client_loop();
    }
    b <@ Env.final();
    return b;
  }
}.
```

PCR Protocol

```
proc client_loop() : unit = {
  var cnt : int; var tag : tag;
  var qry_opt : elem option;
  var not_done : bool <- true;
  while (not_done) {
    qry_opt <@ Env.get_qry();
    cv <- cv ++ [cv_got_qry qry_opt];
    if (qry_opt = None) {
      not_done <- false;
    } else {
      tag <@ Or.hash((oget qry_opt, client_sec));
      cnt <@ tp_count_tag(tag);
      cv <- cv ++
            [cv_query_count(oget qry_opt, tag, cnt)];
      Env.put_qry_count(cnt);
    }
  }
}
```

Adversarial Model

- We are modeling what is called *semi-honest* or *honest-but curious* security
- In this model, the Adversary is given access to a given protocol party's *view*—the party's data—but it is not allowed to modify that data
- The Adversary is also given access to the `hash` procedure of the random oracle — this is different from having access to its map
- The Real and Ideal games for each protocol party are parameterized by the Adversary
  - The Adversary tries to learn more from the protocol's view plus the `hash` procedure's view of the random oracle than it *should*
- At the end of the games, the Adversary returns a boolean judgement, trying to make the probability it returns `true` be as different as possible in the Real and Ideal games

Real Games

- The Real Games for the Server, Third Party and Client are formed as specializations of **`Protocol`**
- For a given party, we define the module type **`ADV`** of Adversaries for that party
  - In calls to the Adversary, the party's current view is supplied
- The Real Game **`GReal`** is
  - parameterized by **`Adv : ADV`**
  - defined by giving **`Protocol`** an environment **`Env`** made out of **`Adv`**

Example: Adversary for Server

```
module type ADV(O : RO.OR) = {
  proc * init_and_get_db(view : server_view) :
    db option {O.hash}
  proc get_qry(view : server_view) : elem option {O.hash}
  proc qry_done(view : server_view) : unit {O.hash}
  proc final(view : server_view) : bool {O.hash}
}.
```

- Adversary can do hashing when deciding which database and queries to choose
- Queries are chosen one by one — *adaptively*
- `qry_done` is called with server view, which does not include the count for the query
- Each time the Adversary is called, it can do hashing to try to increase its knowledge

Example: Real Game for Server

```
module GReal(Adv : ADV) = {
  module Or = RO.Or
  module A  = Adv(Or)

  module Env : ENV = {
    proc init_and_get_db() : db option = {
      var db_opt : db option;
      db_opt <@ A.init_and_get_db(Protocol.sv);
      return db_opt;
    }

    proc get_qry() : elem option = {
      var qry_opt : elem option;
      qry_opt <@ A.get_qry(Protocol.sv);
      return qry_opt;
    }

    proc put_qry_count(cnt : int) : unit = {
      A.qry_done(Protocol.sv);
    }
```

Real Game for Server

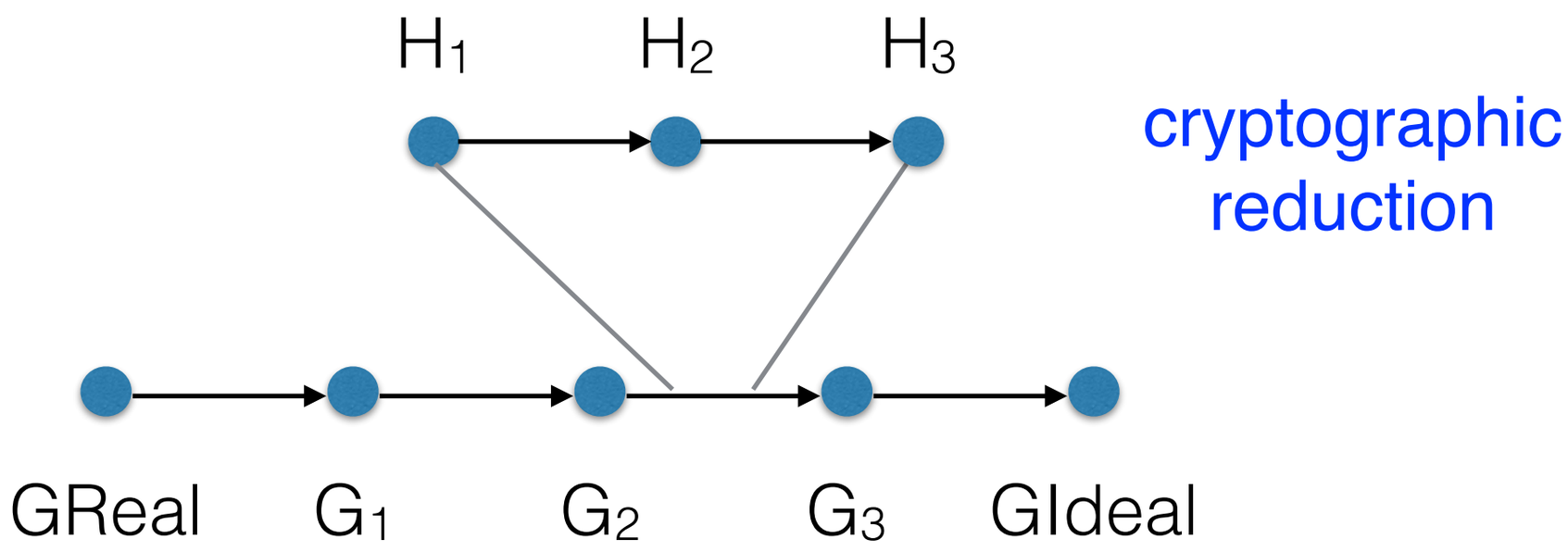
```
  proc final() : bool = {
    var b : bool;
    b <@ A.final(Protocol.sv);
    return b;
  }
}

proc main() : bool = {
  var b : bool;
    b <@ Protocol(Env).main();
    return b;
  }
}.
```

Ideal Games

- A party's Ideal Game is also parameterized by a Simulator (in addition to the Adversary)
- Simulator's job is to convince the Adversary it's interacting with the real game: it must simulate the party's view and the hashing function's view of the random oracle state
- Because we are working information-theoretically, when assessing the information leakage from the Ideal Game to the Simulator (and thus Adversary), we don't have to scrutinize its Simulator
    - It can't learn more about the database or queries by brute force computation
- In fact, in our EasyCrypt security theorems, the Simulators are existentially quantified

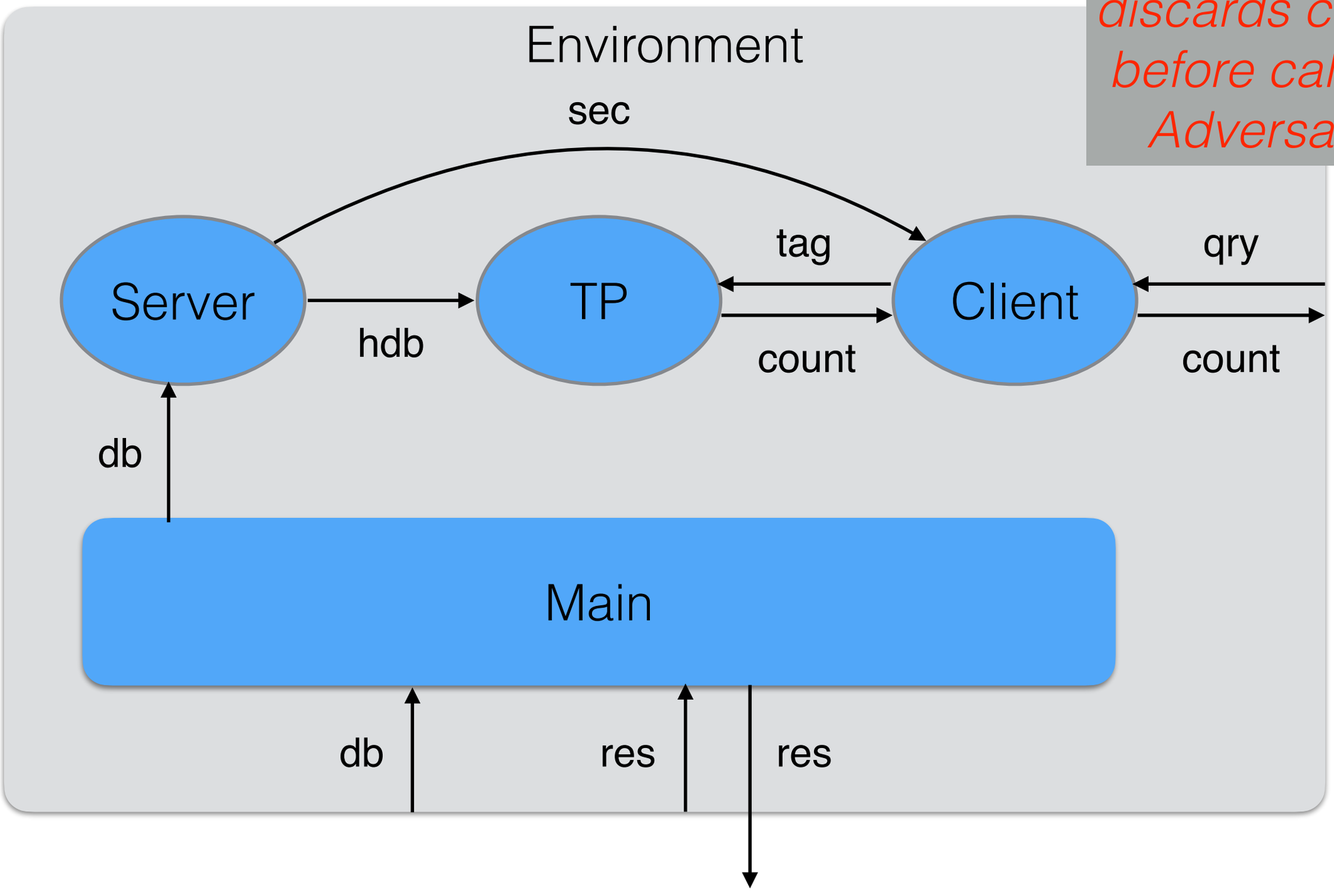
Two Dimensional Sequences of Games

- When proving security against a protocol party, we use EasyCrypt's pRHL and ambient logics to connect the party's Real and Ideal Games via a sequence of games
  - Upper bound on distance between source and target games is sum of intermediate transitions' upper bounds
- We can prove a game transition using a previously proved sequence of games



Reminder: Real Game for Server

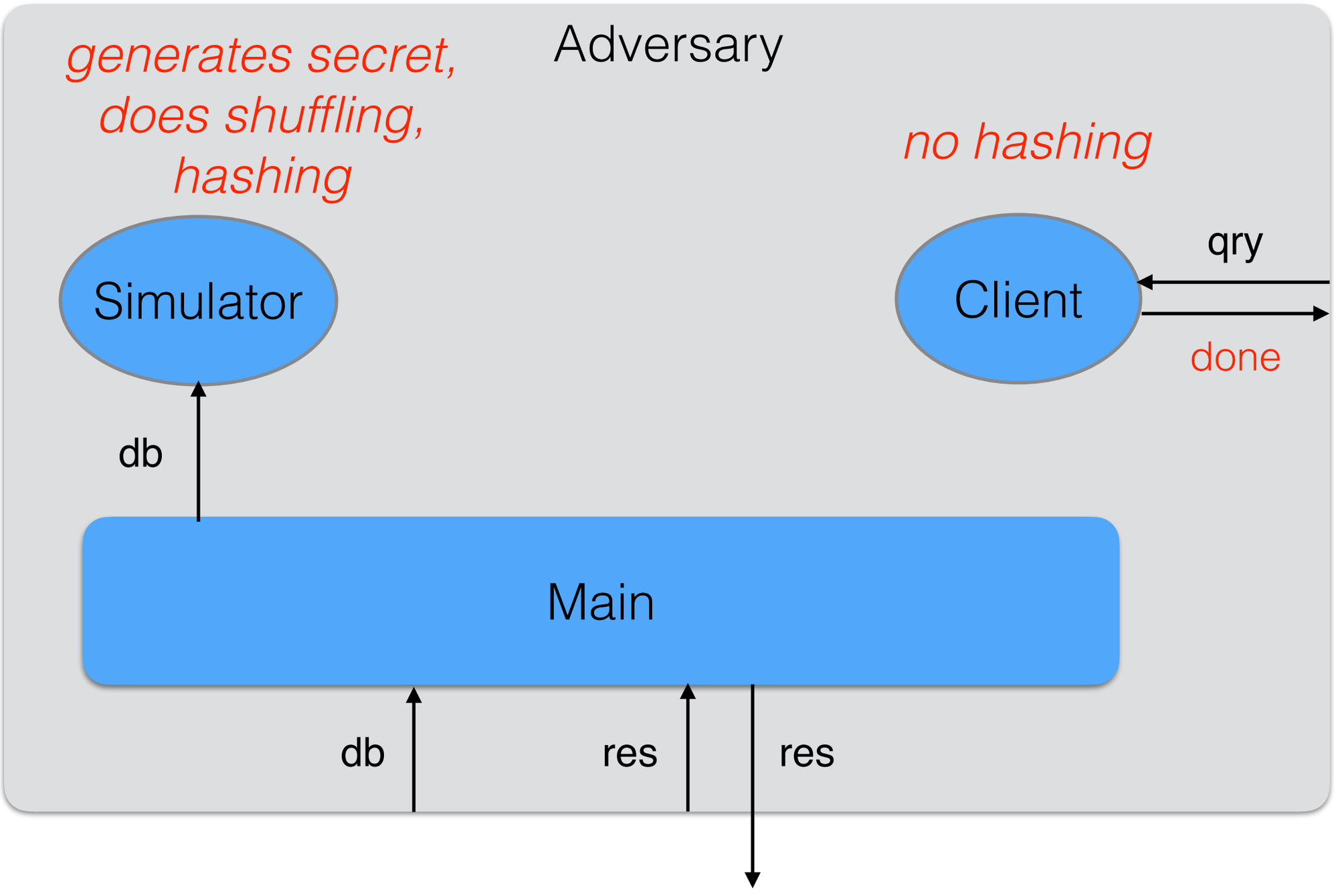
Environment discards count before calling Adversary



Real Game for Server

- What (if anything) can the Server learn about the queries and their counts?
- We formalize this by asking what can be learned from the Server views that are passed to the Adversary — plus the ability to run the `hash` procedure of the random oracle
  - We can think that each time the Adversary is called, the Server is woken up
- To answer and prove this, we need to formalize an Ideal Game

Ideal Game for Server



Ideal Game for Server

- The Simulator doesn't directly learn anything about the queries, and so the Server views it simulates can't convey anything about them either
- And the query loop doesn't modify the random oracle, so experimentation with the random oracle won't learn anything either
- But because the Server is woken up each iteration of the query loop, the Server does learn the number of queries

Proof of Security Against Server

- We are able to prove perfect security: Real/Ideal games equally likely to return true:

```
lemma GReal_GIdeal :

  exists (Sim <: SIM{GReal, GIdeal}),

  forall (Adv <: ADV{GReal, GIdeal, Sim}) &m,

  Pr[GReal(Adv).main() @ &m : res] =

  Pr[GIdeal(Adv, Sim).main() @ &m : res].
```

- The only challenge is dealing with the redundant hashing performed by the Client in the Real but not the Ideal Game
- We remove it using a variation of a technique due to Benjamin Grégoire

Redundant Hashing

```
module type HASHING = {
  proc hash(inp : input) : output
  proc rhash(inp : input) : unit
}.

module type HASHING_ADV(H : HASHING) = {
  proc * main() : bool {H.hash H.rhash}
}.
```

Two implementations of `HASHING`, both built from a random oracle `O`:

- `NonOptHashing` (``non optimized hashing''), in which `rhash` hashes its input, but discards the result
- `OptHashing` (``optimized hashing''), where `rhash` does nothing

Redundant Hashing

```
module GNonOptHashing(HashAdv : HASHING_ADV) = {
  module H  = NonOptHashing(Or)
  module HA = HashAdv(H)
  proc main() : bool = {
    var b : bool;
    Or.init(); b <@ HA.main();
    return b;
  }
}.
```

```
module GOptHashing(HashAdv : HASHING_ADV) = {
  module H  = OptHashing(Or)
  module HA = HashAdv(H)
  proc main() : bool = {
    var b : bool;
    Or.init(); b <@ HA.main();
    return b;
  }
}.
```

Redundant Hashing

```
lemma GNonOptHashing_GOptHashing
        (HashAdv <: HASHING_ADV{Or}) &m :
  Pr[GNonOptHashing(HashAdv).main() @ &m : res] =
  Pr[GOptHashing(HashAdv).main() @ &m : res].
```

- Proof intuition: redundant hashing can be put off until it's superseded by `hash` or no longer necessary
  - Proof uses EasyCrypt's `eager` tactics

- To use in Server proof, we define a concrete adversary `HashAdv` in such a way that the left side of the gap in the sequence of games proof can be connected with `GNonOptHashing(HashAdv)`, and `GOptHashing(HashAdv)` can be connected with the right side of the gap

Next Class: Third Party and Client Ideal Games and Proofs