OpenSSLのバグを見つけた話

菊池正史@株式会社レピダム

TLSタイムライン2014

- 2月
  - goto fail;
  - goto fail;
- 4月
  - Heartbleed
  - みんなOpenSSLのバグを探し始める

TLSタイムライン2014

- 4月

TLSタイムライン2014

- 4月
  - LibreSSL
  - CCS Injection発見
- 6月
  - OpenSSLリリース。たくさんのバグ修正

CCS Injection脆弱性とは

- CCS = Change Cipher Spec
  - TLS/SSLのメッセージ
  - ここから暗号を変えますよ！
- CCS Injection脆弱性
  - 中間者がCCSを挿入すると
  - OpenSSLが気にせずに受理しちゃって
  - 変なタイミングで暗号が変わって
  - 中間者が通信を完全に解読できる

脆弱性を見つけた話

- みんながバグ探し競争してるので効率的に探したい
- Coqもどこかで使いたい
- バグのありそうなモジュールを経験的に決めうちする
- 意味不明なコードを理解する足がかりが欲しい

Coqとは？

- 定理証明器
  - Inriaで開発された
  - 命題を自分で書いて
  - 証明も自分で書く
  - 証明が正しいことは自動で検証してくれる
  - 少しは自動証明もやってくれる
- http://www.iij-ii.co.jp/lab/techdoc/coqt/

Coqでプログラミング

- Coqはプログラム言語でもある
  - 無限ループは書けない
  - IOはできない
- 簡単なプログラムを対話的に実行するだけなら、Coqの対話環境でできる
- 本格的なプログラムを作るには、2種類のアプローチがある

Extractionを使ったプログラミング

- CoqのExtractoin機能を使うと
- OCamlのプログラムを出力してくれる
- 他のOCamlのプログラムとリンクすることで、普通のアプリケーションを作ることも可能
  - 無限ループみたいなのは一番外側のループをOCamlで書く等

Coq上のデータでプログラムを表現

- Coq上では目的プログラムがデータとして見えるので、どんなプログラムでも書ける
- セマンティクスを与えないと証明も何もできない
- この方法は難しいので、まだ発展途上
  - C言語や機械語を埋め込むのが普通

CoqをつかってTLSを実装したい

- 週末の趣味の時間を作ってできるぐらいの簡単なものを作りたかった
- GWの間に作る予定だった
- Extractionを使ってメインの状態機械を実装したOCamlのコードを出力して
- 外側のループやソケットIOや暗号のプリミティブなんかはOCamlで書く
- OpenSSLのコードリーディングの助けにもしたい

バグのありそうなモジュール

- TLSパケットのパーザ
  - ここはバグの宝庫
  - 死ぬほどバグが残ってるのは明らか
  - Heartbleedもここのバグ
  - みんなここのバグを探してるから競争率が高い
  - TLSの知識がなくてもそれなりにバグが見つかるからハードルも低い

バグのありそうなモジュール

- TLSパケットのプリンタ
  - パーザの逆の動作をする
  - ここにバグを入れるほうが難しい
  - バグがあると他のTLS実装と通信したときに、真っ先にエラーになるはず

バグのありそうなモジュール

- ハンドシェークのステートマシン
  - 巨大なswitch文で書かれてるけど、普通のステートマシンなので、読みにくいわけではない
  - TLSのステートマシンは分岐がほとんどない一本道なので、バグの入る余地が少なそう
  - そもそも、バグがあったら他の実装と通信できない？

バグのありそうなモジュール

- ソケット入出力とバッファリング
  - ここは実装したことがないと分かりにくいところ
  - 非同期IOのサポートとかのAPI設計やパーザのモジュールがどのようにバッファにアクセスするかとかがめんどくさい
  - 暗号が切り替わる瞬間も考えないといけない

バグのありそうなモジュール

- ASN.1
  - パーザと並ぶバグの宝庫
  - ASN.1のパーザは何故かすぐにバグるし、解釈もバグる
  - でも、よく知らないので今回はパス

バグのありそうなモジュール

- 暗号モジュール
  - テストしやすいので普通はバグはない
  - ちょっとバグってると盛大にぶっこわれるのですぐに気づく
  - 暗号学的にどうなのかというのは専門家の考える仕事なので気にしなくてもよい

Coqが使えそうなところ

- パーザ・プリンタの対
  - Coqを使えば、かなり信頼できるのが作れる
  - 新しいTLS実装を作りたいときのたたき台にするのは面白い
  - ただし、既存の実装を理解する用途には使いにくい
  - パージングの戦略とかバッファへのアクセス方法とか合わせないといけない

Coqが使えそうなところ

- ステートマシン
  - たぶん、巨大なswitch文で書いてもあんまり変わらない
  - ほとんど一本道なので面白くない
  - たぶんバグはないとみんな思ってる

Coqが使えそうなところ

- 入出力とバッファリング
  - 暗号が切り替わる部分でバッファをどう操作すればいいのか実装したことがないとわからない
  - 暗号の切り替わりタイミングはステートマシンが遷移するところなので、ステートマシンと同期して動かさないといけない
  - Coqで書いたコードと比較しやすそう！

バッファリング周りのコード

- https://github.com/openssl/openssl/blob/master/ssl/s3_pkt.c#L139
- たぶんこの辺
- ぱっとみ、ソケットからバッファにnバイト読んでくる関数なんだと思われる
- 読んでみても何がなんだかさっぱり理解できない
- 必要なバッファとアクセスのしかたを人間が読める形で書いて理解しないと無理

よくあるバッファの作り方

- 固定サイズ(capacity)の配列
- readerが次に読み出す位置(r_pos)
- writerが次に書き込む位置(w_pos)
- 0 <= r_pos <= w_pos <= capacity
- というような複雑なデータ構造は人類には難しいので、ただのリストでいい。
- r_posからw_posまでの内容をリストで持つ

バッファの定義

Definition Buffer := list UInt8.

- バッファ型はUInt8のリストとする
- バッファの操作は後ろにバイト列を追加するのと、前からバイト列を取り出すものがある

後ろにバイト列を追加

```
Definition Enqueue (b : Buffer) (x : list UInt8)
  : Buffer := b ++ x.
```

- 単にリストを結合するだけ。自明。

前からバイト列を読み出す

- 失敗する場合分けがあるので人類には難しい
- このような部分関数は述語で書いたほうが分かりやすい

前からバイト列を読み出す

```
Definition DequeueSpec1 b n a b' :=
    b = a ++ b' ∧ length a = n
```

- バッファbの先頭からnバイト読み出した結果が、バイト列aであり残りのバッファがb'であるという述語
- 引数nはlength aに等しくないと真にならないのに渡す必要あるの？

前からバイト列を読み出す

```
Definition DequeueSpec b a b' :=
    b = a ++ b'.
```

- 定義は分かりやすくなった。Enqueueと対称になってるのもかっこいい
- 関数として解釈すると、バッファbとバイト列aを受け取って、bの先頭がaにマッチしたときに、残ったバッファを返す関数

この定義の便利なところ

Parameter HandshakeMsg: Type.

Parameter HSMsgPrinter

: HandshakeMsg -> list UInt8.

- ハンドシェークメッセージを表す型と
- それをバイト列にプリントする関数が与えられたとする

この定義の便利なところ

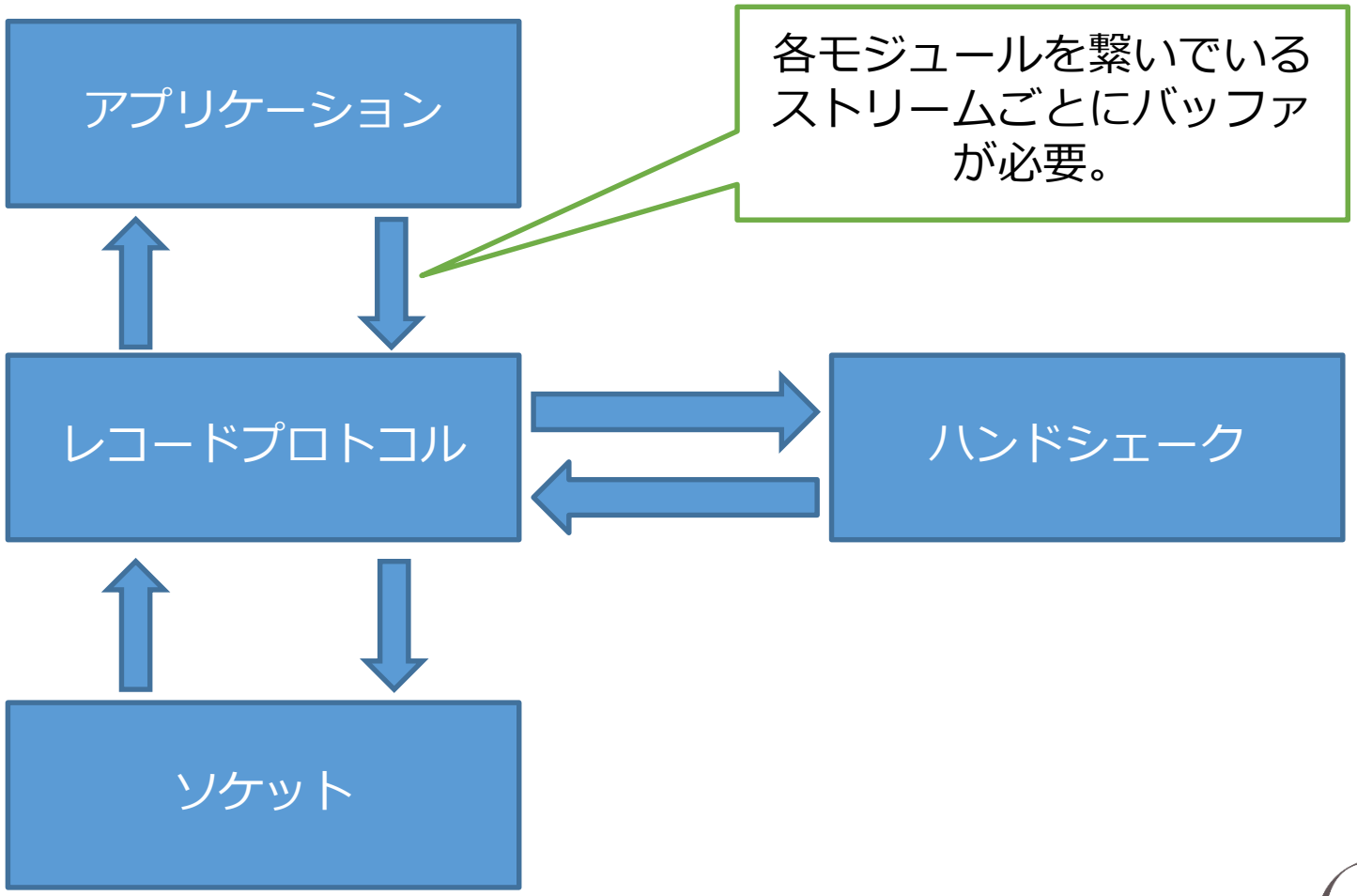
```
Definition DequeueHS b b' := ∃ hs,
  DequeueSpec b (HSMsgPrinter hs) b'.
```

- ハンドシェークメッセージhsで、プリントした結果がバッファの先頭にある場合は、それをバッファから取り出す関数を定める述語
- すなわち、このバッファを使ったパーザ関数を簡単に特徴づけられる

バッファ付きステートマシン

- ステート変数と6つのバッファを持った状態を定義する
- なぜ6つかというと・・・

6つのバッファ



6つのバッファ

- socketIn ソケットから読み出したバイト列
- socketOut ソケットに書き込む
- appInBuffer アプリが受け取る
- appOutBuffer アプリが送り出す
- handshakeInBuffer HSサブプロトコルの入力
- handshakeOutBuffer HSサブプロトコルの出力

可能な状態遷移を列挙する

- 例えば
- socketInBufferからハンドシェークレコード r を読み出して
- 中身のフラグメント部分をhandshakeInに追加する遷移が可能であるという述語は
- DequeueSpec sIn (toOctets r) sIn'∧
- EnqueueSpec hsIn (fragment r) hsIn'

一番複雑な状態遷移

- こんな感じで状態遷移を列挙していくと
- ChangeCipherSpecというレコードがsocketInBufferの先頭にある場合が一番複雑であることが分かる
- 暗号が切り替わるのでバッファをクリアしないといけない(なので、状態の中に暗号の状態もいれないといけないのがTODOだった)

一番複雑な状態遷移

- ここまでたどり着くまで3日
- というのも、TLSを実装したことがあったから
- やっぱり一番複雑なところは既存の実装を見て自分の考えが正しいかどうか確認したくなるよね
- というわけで、OpenSSLがChangeCipherSpecをどのようにハンドルしてるか確認してみた！

OpenSSLにおける実装

- 上で考えてたようなめんどくさい状態管理とは全く関係なく、最初から完全にぶっこわれていた
- ここからが大変
- ぶっこわれている処理を使った実証コードを書いたり
- パッチを書いたり
- 報告したり色々

実証コードの作成

- rubyで300行くらい
- 暗号プリミティブはopensslライブラリを使用
- 二日ほどで完成
- 壊れた状態でハンドシェークを完了させるために、細かな工夫が必要
- 詳細はImperial violetってブログに書いてある

バグ修正の話

- バグ修正は大変そうだったので最初は全くやる気がなかった
  - TLSを直すのはまあできる
  - DTLSも直さないといけないはず
    - DTLSのコードはTLSのをコピペして
    - UDP固有のコードをごっそりと追加している
    - OpenSSLのコードの中でも難解な部分
  - そもそもDTLSなんて全く知らない

バグ修正の話

- ようやくやる気を出してDTLSのコードを読んだら、なぜかすでに直っていた。
  - DTLSの場合はパケットの順番が入れ替わる可能性があるので、CCSが先に来てしまうことも容易にある
  - 脆弱性にはならないと判断されて調査されてなかった
  - この修正をコピペするだけで直るはず

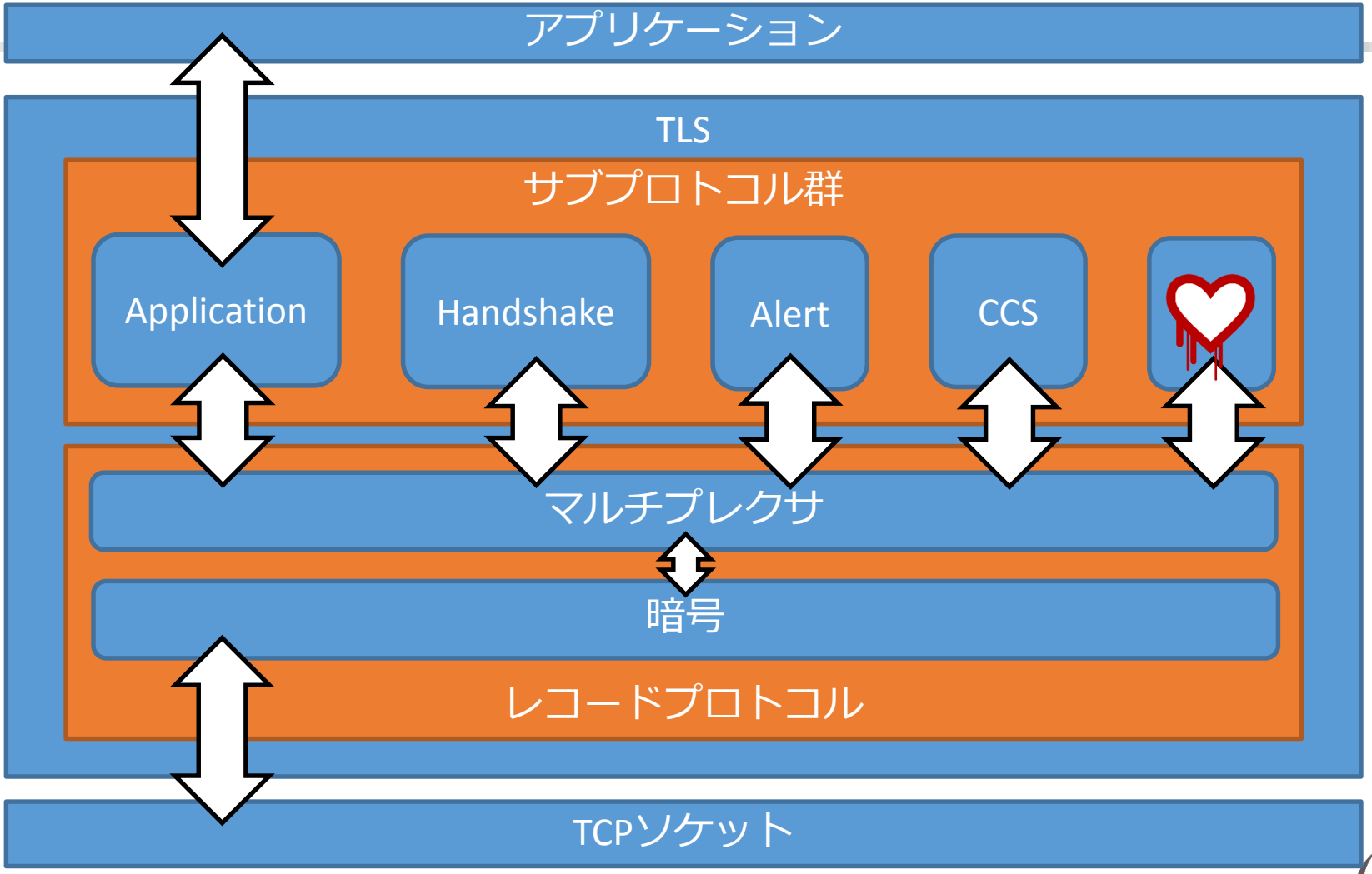
バグ修正の話

- DTLSではCCSを受け取っていいタイミングかどうかのフラグを追加して直していた
- TLSにそのフラグを追加する案はだめだった
  - 構造体にフィールドを追加するとABI互換性がなくなる
  - OpenSSLみたいな古い設計のライブラリは全く考慮されて作られてない
  - 結局、すでにあるフィールドの空いてるビットにいれることになった

まとめ

- TLSぐらいの複雑さのプロトコルをCoqでいじってみようとするときは先に普通に実装した経験がないとつらいと思う
- 結局、Coqのコードは100行ぐらいしかできなかった
- 脆弱性対応は大変だった
- あまりのショックにGW中はずっとマインクラフトしかできなかった

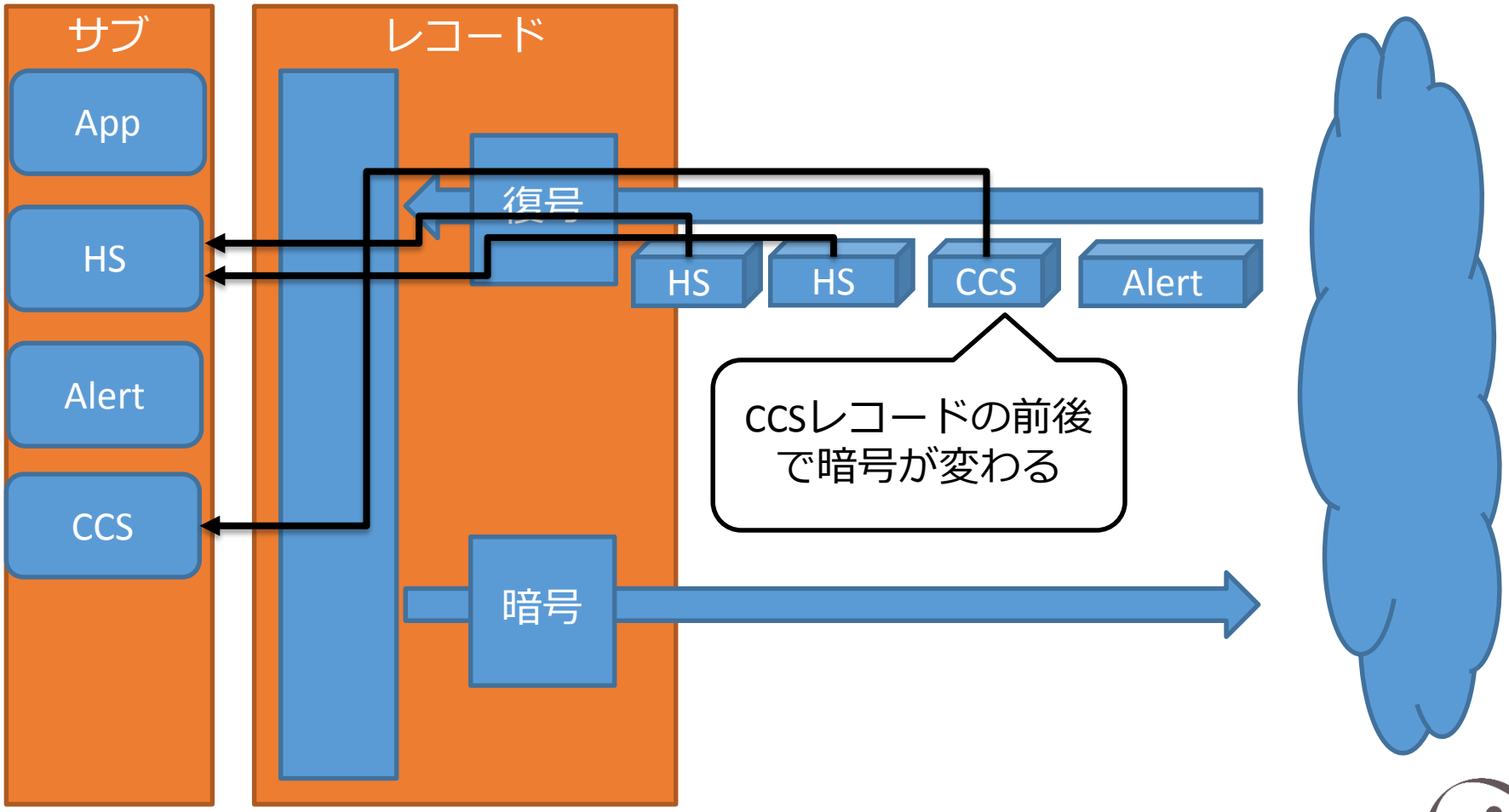
TLSアーキテクチャ



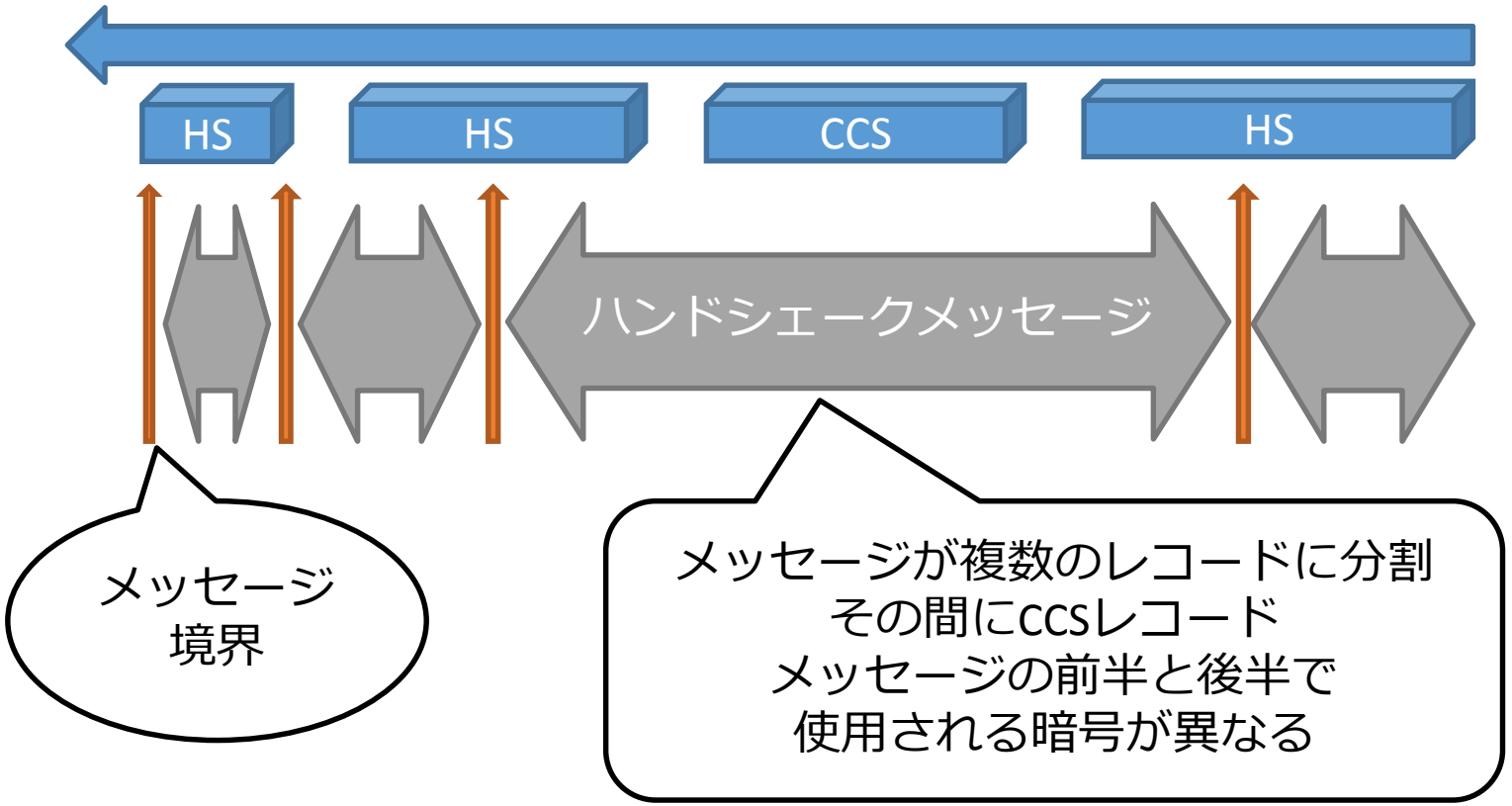
レコードプロトコル

- サブプロトコルの通信をレコードという単位にカプセル化
- サブプロトコルからは独立したストリームに見える
  - メッセージ境界などは保存されない
    - ClientHelloメッセージが複数のレコードに分割されるかもしれない
    - ひとつのレコードに複数のメッセージが入るかもしれない

レコードプロトコル



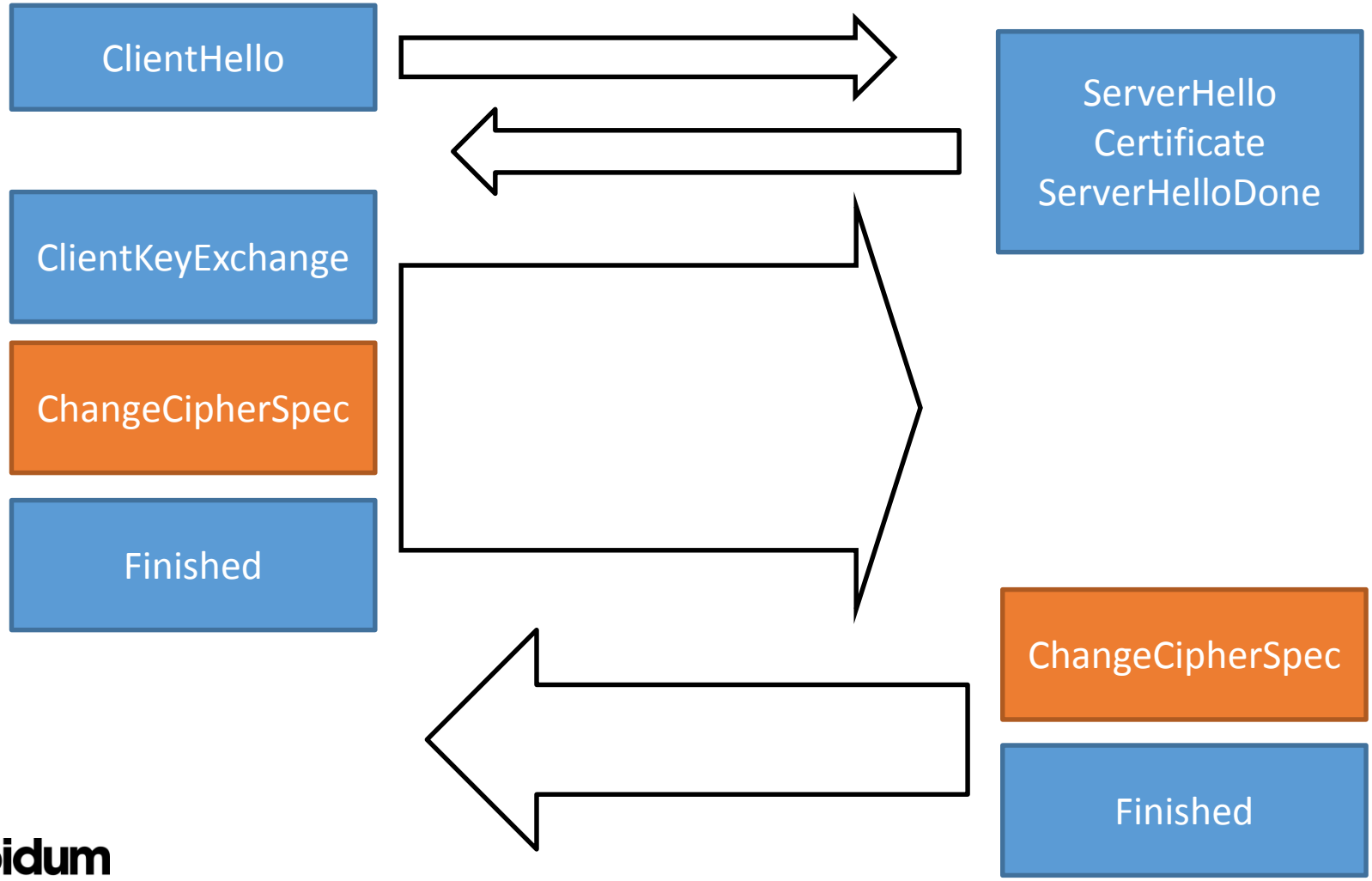
フラグメントの問題



CCSとメッセージ境界

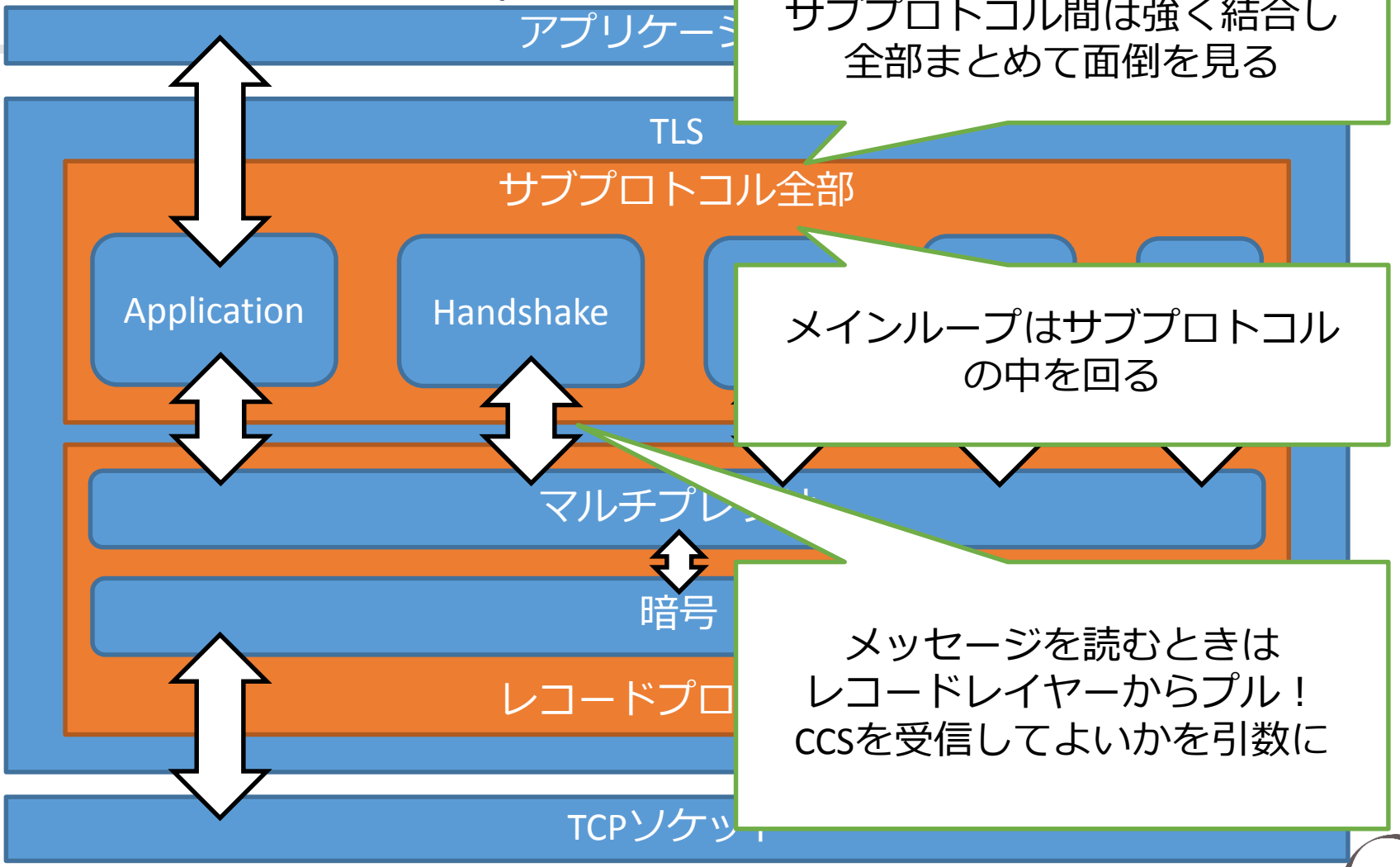
- 分割されたメッセージの間にCCSがあってはならない
- CCSを受理するときには全てのサブプロトコルがメッセージ境界にぴったり合っていることを確認する必要がある
- RFCには全く書いてない
- CCS処理はサブプロトコル間の複雑な同期が必要！

ハンドシェーク



lepidum
https://lepidum.co.jp/

よくある設計(その１)

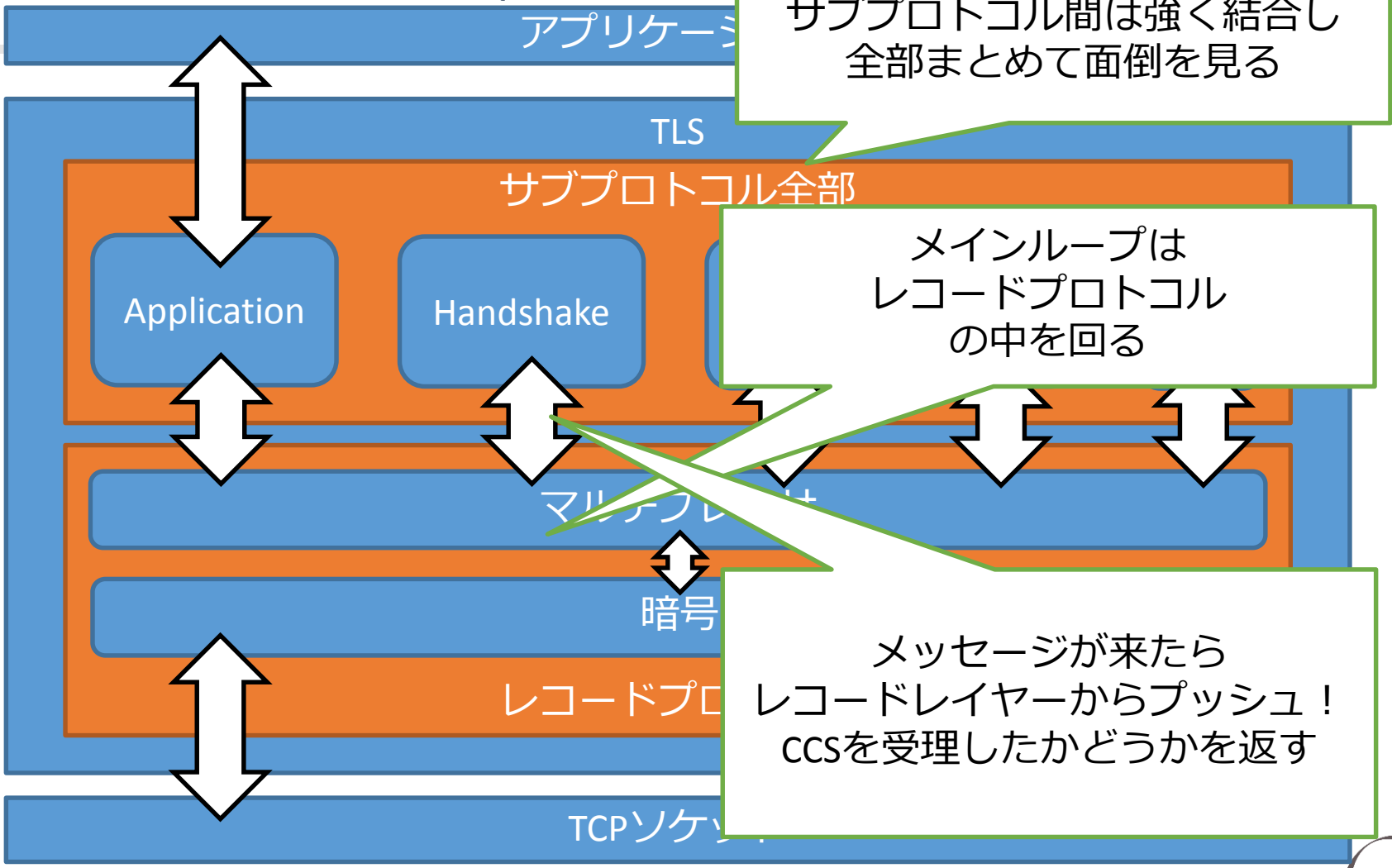


サブプロトコル間は強く結合し
全部まとめて面倒を見る

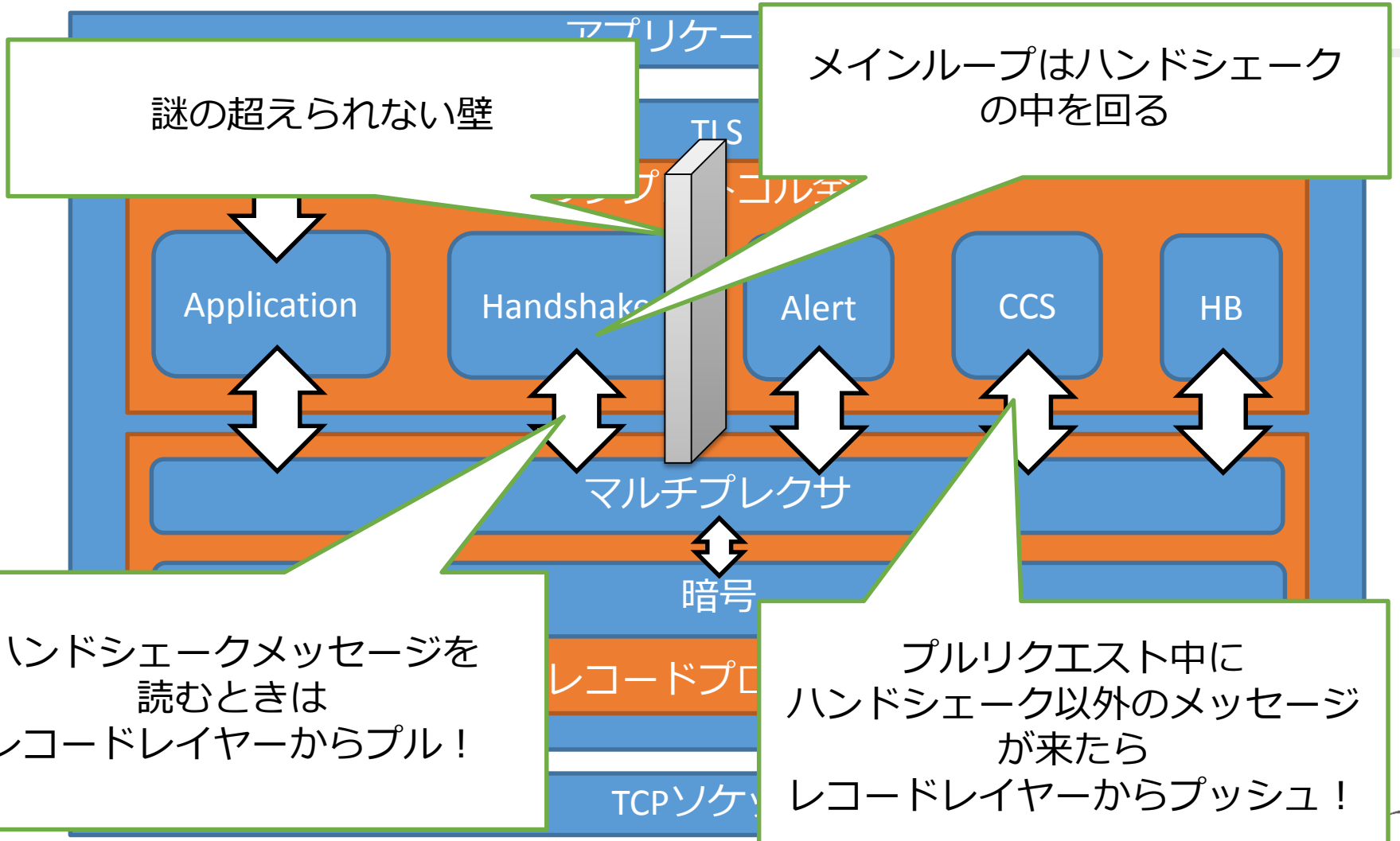
メインループはサブプロトコル
の中を回る

メッセージを読むときは
レコードレイヤーからプル！
CCSを受信してよいかを引数に

よくある設計(その２)



よくわからない設計！



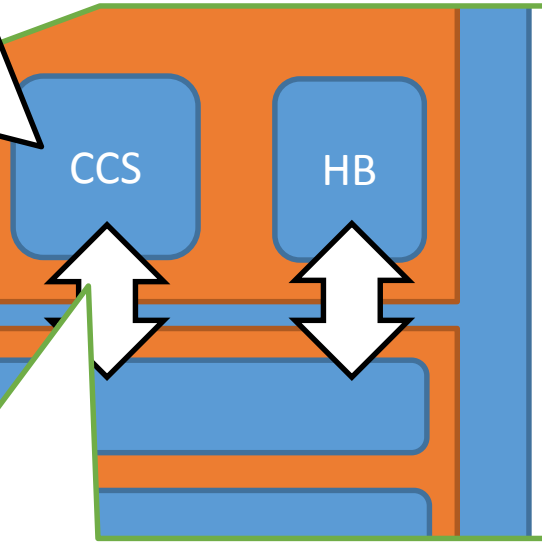
lepidum
https://lepidum.co.jp/

よくわからない設計！

CCSがプッシュされて来たら
何も考えずに受理！
暗号を変える！

あと、CCSを受け取ったよ
フラグを立ててみる

ンループはハンドシェーク
の中を回る

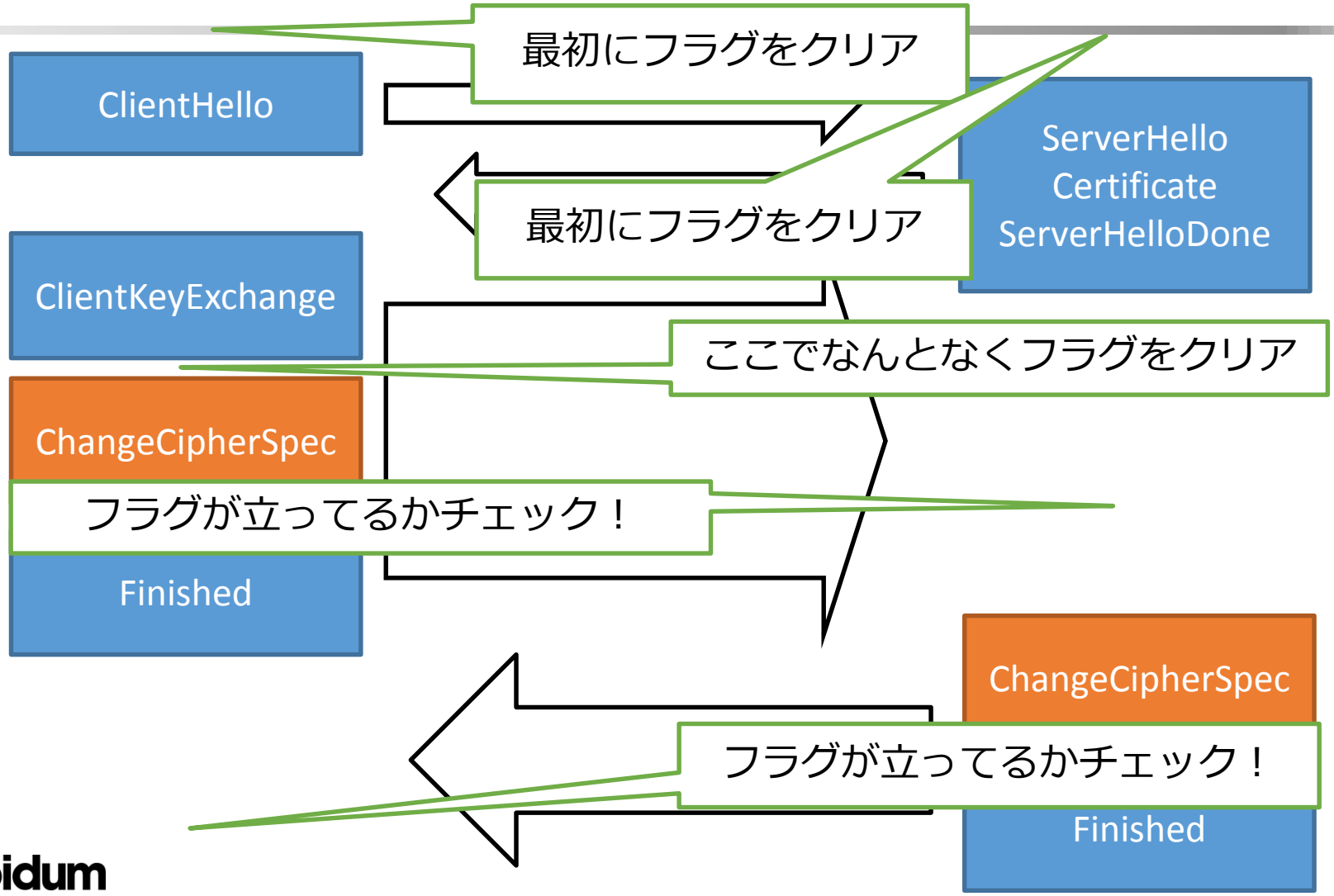


プルリクエスト中に
ハンドシェーク以外のメッセージ
が来たら
レコードレイヤーからプッシュ！

レコードレイヤーからプル！

TCPソケ

CCSを受け取ったよフラグの管理



lepidum
https://lepidum.co.jp/

まとめ

- いつでもccsを受理するようになってた
  - ccsを受信したよフラグで管理できると考えてたっぽい
- クライアントはccs受信フラグを途中でクリアする
  - クリアする前にccsを受け取ってても忘れる！
- サーバはccs受信フラグをクリアしない
  - フラグチェック前ならいつ受信しても同じ！