加速器制御システム用コルーチンベース非同期ブリッジ

v0.3

Alex Gimenez – alexandre.Gimenez@riken.jp

概要

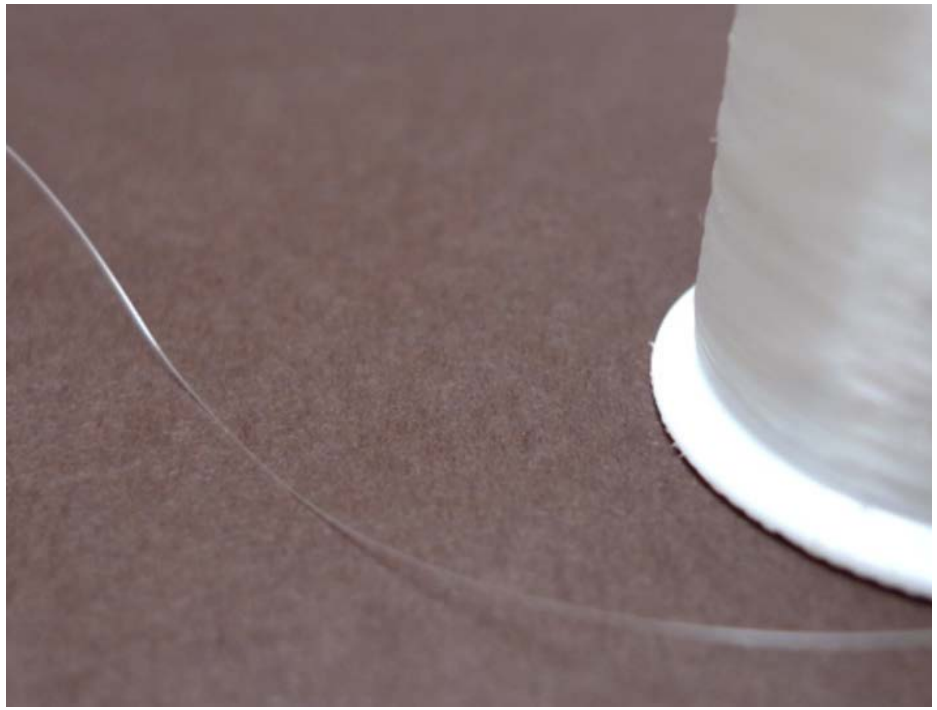
- 目的
- 背景
- 既存ソフトウェアとの統合
- Asynchronous Bridgeの構成・動作
- Event Loopとの協働動作
- 結論

目的

- **改善すべき点**
  - 加速器制御ソフトウェア開発には
    - **生産性**
    - **品質**
- **同期プログラミング**
  - 同期処理は**シンプル**で、改善につながる
- **統合** 非同期と同期
  - 非同期ソフトウェアライブラリが豊富にある
  - 新しく作成された同期ソフトウェアは、既存非同期レイヤに統合すべき
  - 統合は複雑で、その統合をしやすくするライブラリを作成した

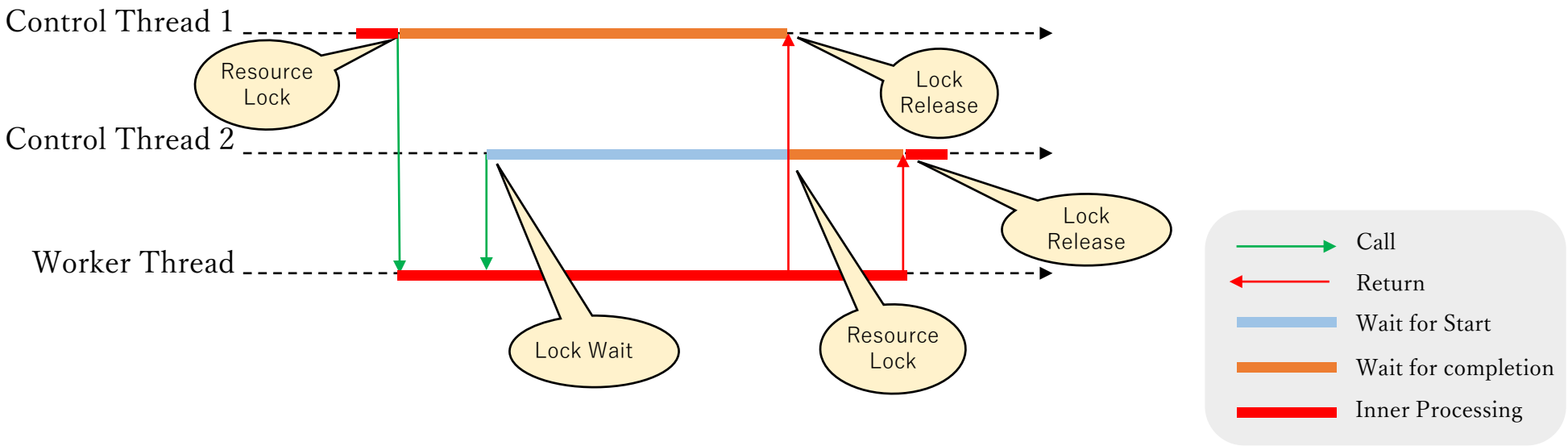
Thread

Fiber



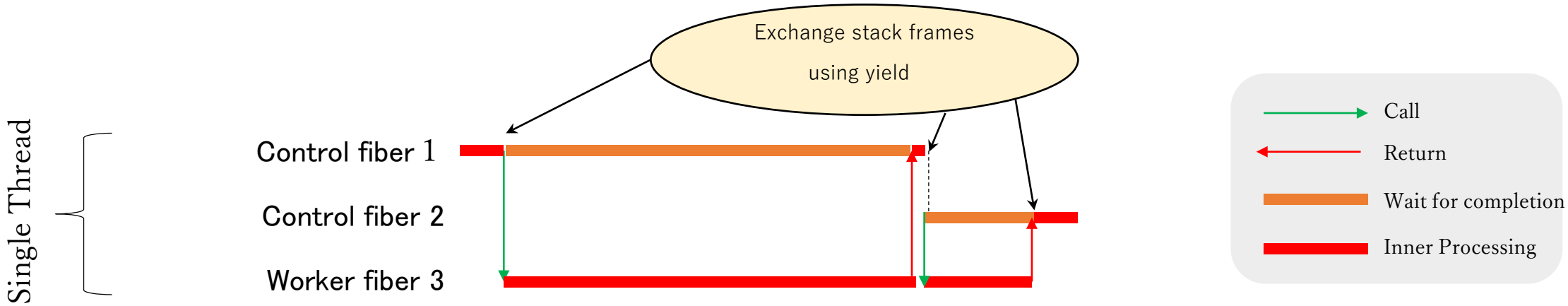
背景・1

- 非同期インターフェイスを同期レイヤで隠蔽することは一般技術
- よく、スレッドを使って隠蔽します
- 尚、スレッドの作成、維持、消滅で、複雑さが上がってしまい、品質に悪影響があり
- さらに、コンテキストスイッチによって、電源無駄遣いも、リアルタイム悪影響も発生



背景・2

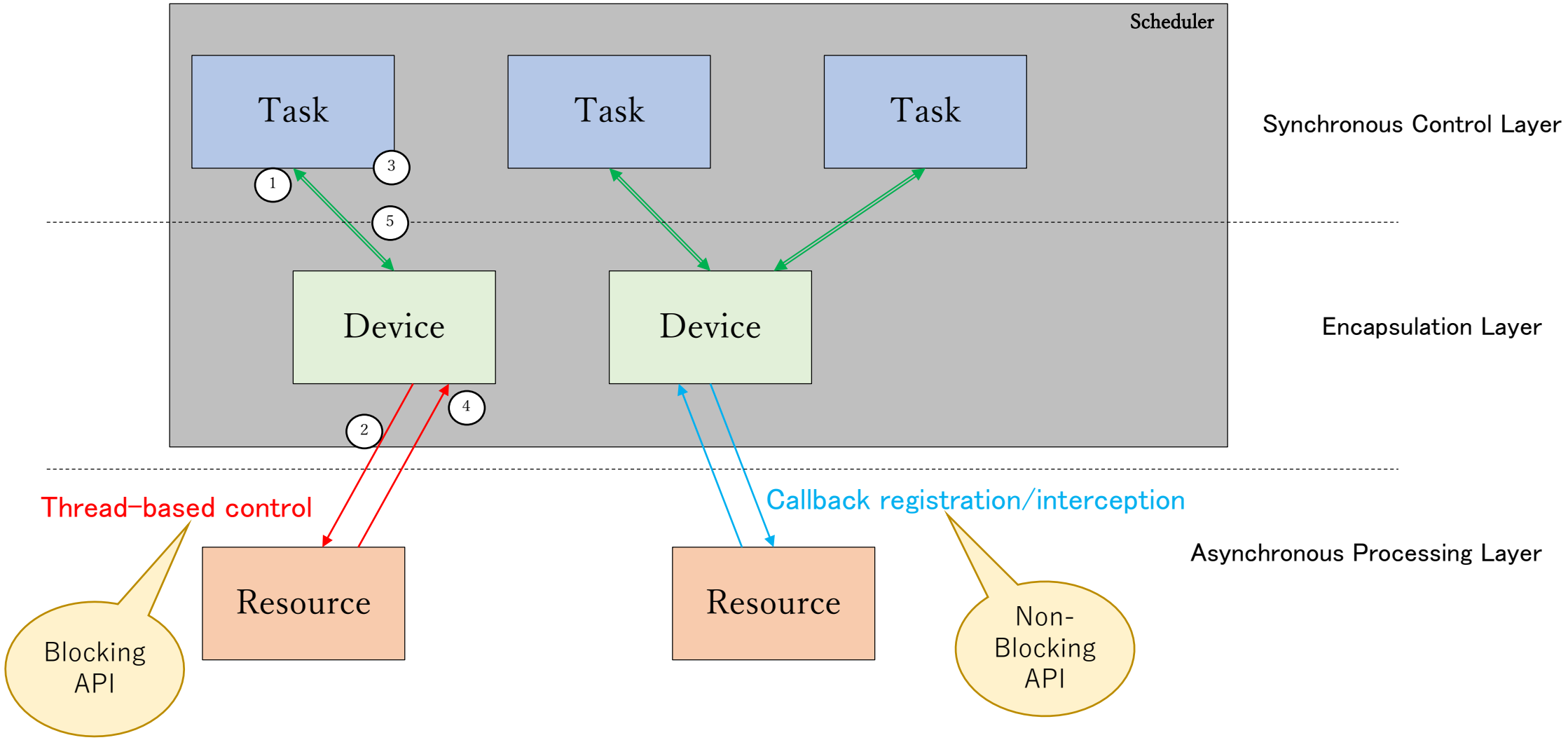
- コルーチン（Cooperative Multitasking, Fiberとも言い）、制御を譲りながら同期プログラムが作成できます。
- コルーチンは斬新ではない
- 尚、通常でもありません
- 最近、コルーチンの規格化動きは増加（Python, C++17も含めて）



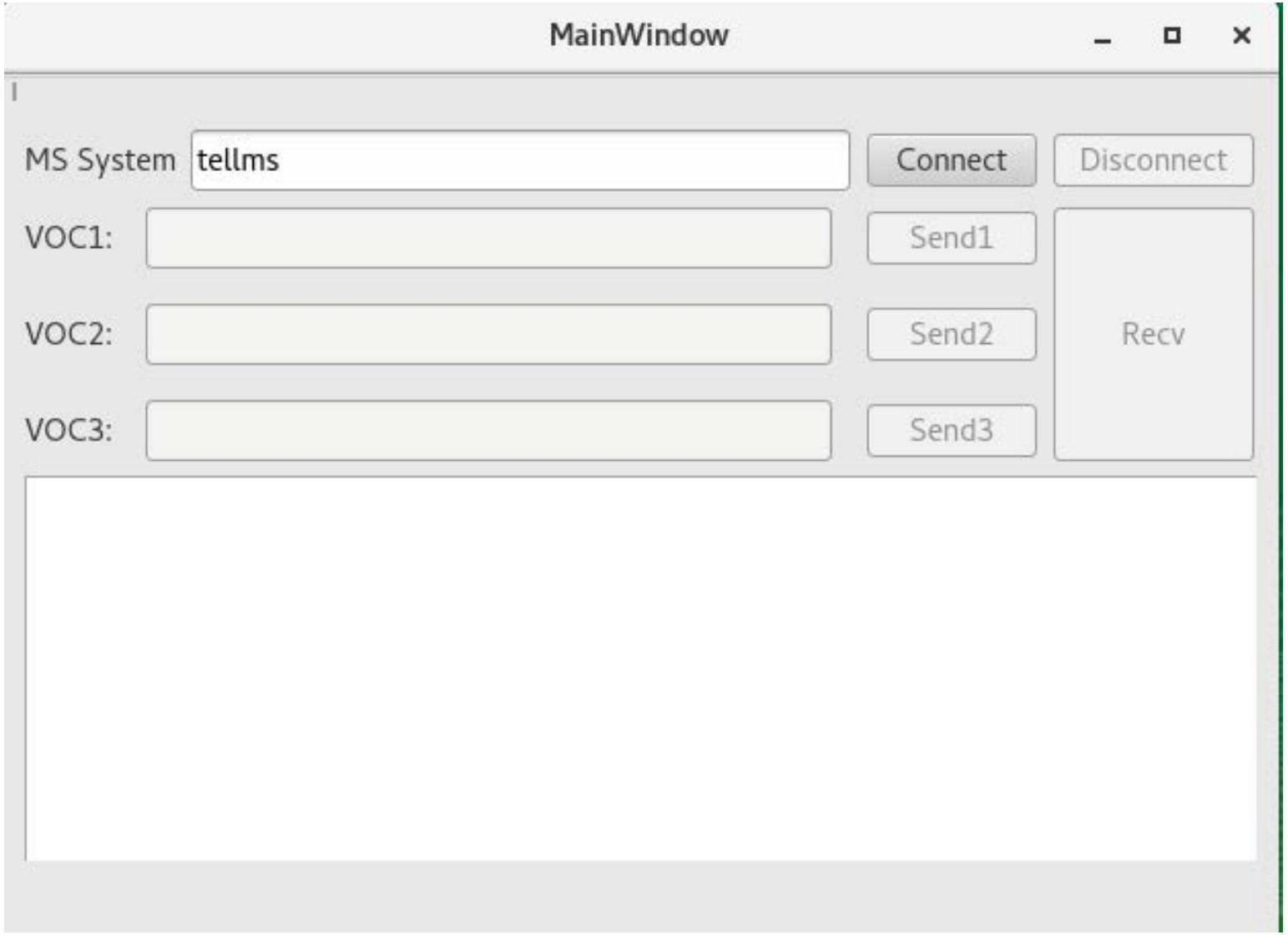
既存ソフトウェアとの統合

- コルーチンからAPIを呼び出すには、呼び出されるAPIにはコルーチン対応性が必須
- 尚、既存APIをコルーチン対応に作り直すことは難しい
- その対策として、「Asynchronous Bridge」という中継レイヤを導入
- Task・Device・Resourceという抽象を使います
- Resourceは下流API
- TaskとDeviceの管理をするSchedulerも利用
- 開発者はDeviceからクラスを継承し、APIを隠蔽
  - 容易データ転送メカニズムを利用できる
- Deviceは、下流APIでのインターファイスによって、適切に制御変換を行う
  - Blocking API ->　スレッド管理
  - Non-Blocking API -> コールバックによって、yield

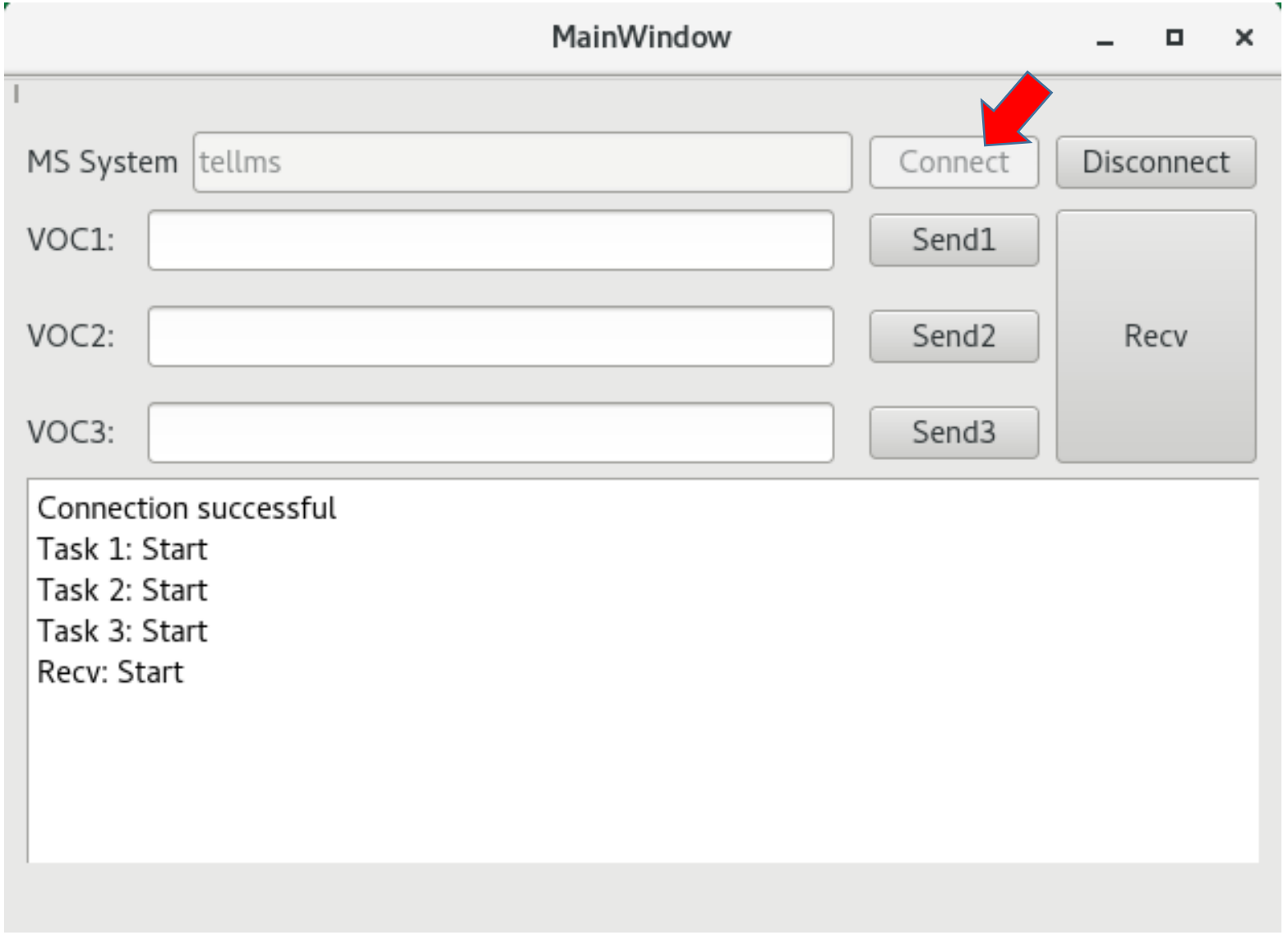
Asynchronous Bridgeの構成・動作



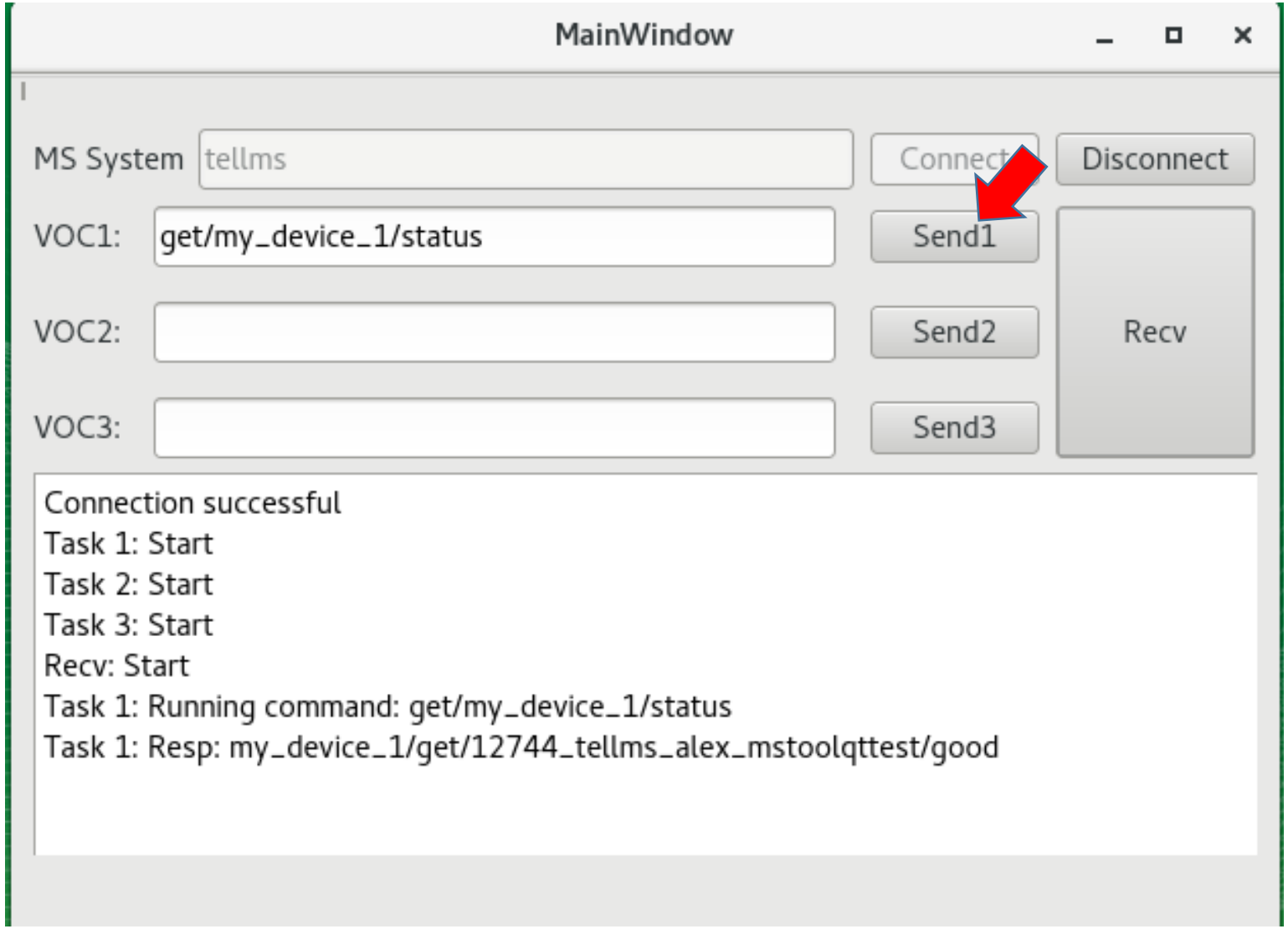
Qt Application - 1



Qt Application - 2

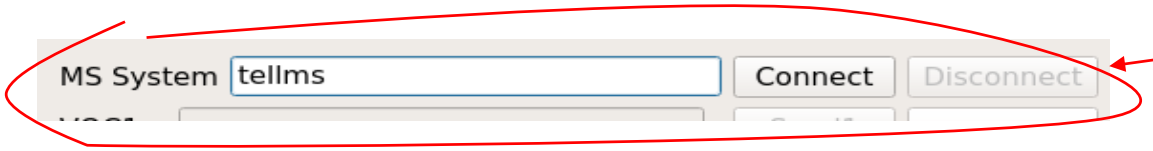


Qt Application – 3



実験：Qtとの統合

- Qtとの統合に、Asynchronous BridgeのDevice抽象を使うと
  - Deviceはいくつかの GUI Widgetを包含
  - Deviceは、Taskからの呼び出しを、GUI Widget プロパティアクセスに変換
  - GUI Widgetからのイベントをタスクへの戻り値に
  - 結果、入力➡処理➡出力　のような制御ができる

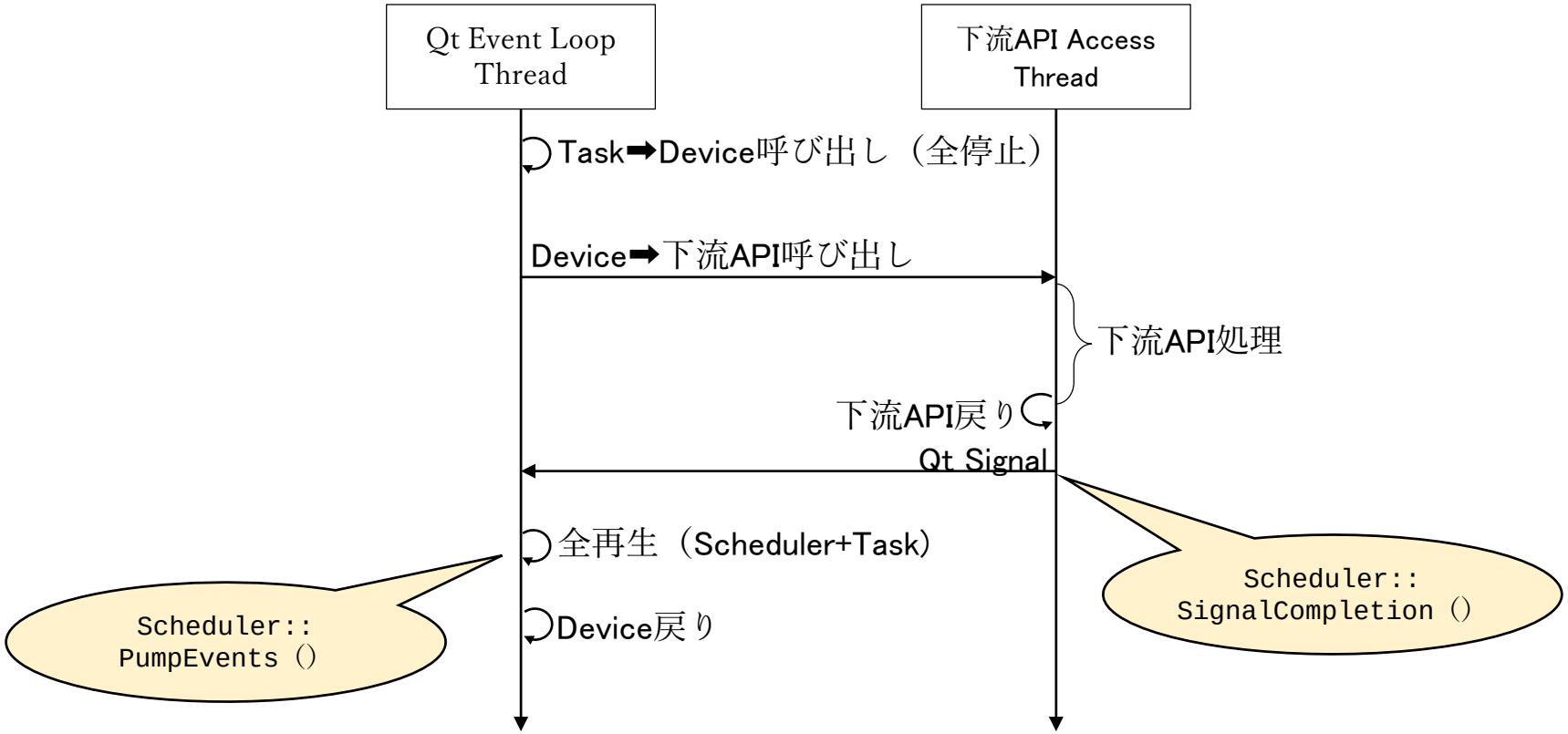


Widget 4個は、一つのDeviceに包含され管理されます。

```
GuiDevice gui_d; //  赤サークルで上記に強調されたGUI Device
ProcDevice proc_d; // ネットワーク処理を行う例Device
while(!d.Finished())
{
    i = gui_d.getInput();
    if( i == Connect  )
    {
         r = proc_d.Connect(  gui_d.GetAddress() ); // text box
         if( r == Success )
         {
             . . .
         }
    }
}
```

Event Loopとの協働動作

- Asynchronous　BridgeとQt Event Loopは同じスレッドをシェア
- Schedulerと全タスク停止の場合、下記の動作が行われます
  - タスクは一時停止　－＞　すべて一時停止だと、Schedulerも一時停止
  - Schedulerは自分のスレッドを持ってないので、誰かに起こされなければならない
  - Schedulerを起こす時、必ず同じスレッドで
- Qtだけでなく、汎用フレームワークとの統合にみ利用できる（Win32で確認）



結論

- **改善すべき点**
  - 人間の脳の動作と計算機内の動作はかなり異なります
  - 生産性と品質の問題の原因の一つ
- **同期プログラミング**で
  - プログラムの動作は、人間の考え方に近づく
- **統合**
  - コルーチンは効果的の同期開発技術
  - Asynchronous Bridgeは、コルーチン対応性を高める道具
    - 統合しやすい
  - それで、同期プログラミングの適用性が上がる