RCI (Remote Class Invocation) ユーザーズマニュアル

初版(v1.0.0) 2018.04.11

■RCI とは

java には RMI(Remote Method Invocation)という遠隔メソッド起動の機能が標準で存在するが、その実行はメソッドに限られる。

RCI は Remote Class Invocation の略で遠隔クラス起動を行う java 言語による独自のプロトコルで、遠隔でクラスを実行できれば、ローカルで実行しているアプリケーションと同じような操作性で様々なことができる。

ネットワークシステムの全てが何をしたいのかと言えば、「サーバ側で何らかのプログラムを実行し、結果をクライアントで受け取りたい」のであり、それらは RPC(Remote Procedure Call)のことで、http もサーバ側で何等かのプログラムを実行し、結果をクライアントで受け取っているので RPC の一種であり、ssh や telnet, ftp やメールの送受信も RPC である。

特に RPC に特化した xml の送受信を行う SOAP（元は Simple Object Access Protocol）というプロトコルや異なる言語間の RPC 用の CORBA という規格もあり、http, https, telnet, ssh, ftp, sftp, SOAP, CORBA などは各種プロトコルであるが、やりたいことは以下のように共通している。

・クライアントとサーバ間で情報交換（通信）したい。

・ユーザ認証をしたい。(http, https はセッションのような拡張概念を使う。http は元々ユーザ認証を前提としていないプロトコルであったので、後付けしても不完全であり、セッションハイジャックのような脆弱性は永遠に解決されない。)

・情報が盗み見られないように暗号化し安全に通信したい。(https, ssh, sftp)

結局のところ暗号化とユーザ認証に対応した高機能な遠隔クラス実行の RCI があれば、http, https, telnet, ssh, ftp, sftp, SOAP, CORBA などの各種プロトコルで行っていた機能を代替できる。

SOAP や CORBA などで RPC を実現するにはサーバやクライアントで接続用の専用のアプリケーションを作らねばならない。wsdl や idl のような難しい文法の疑似言語のようなものでインタフェースを定義し、それからソースを生成するようなことが行われる。これらを利用してシステム化するのは決して簡単なものではない。

単にサーバ側で実行するクラスを用意しておき、設定ファイルで許可するだけで、サーバ側のクラスをクライアント側で遠隔実行できれば開発工数は極めて少なくて済む。インタフェース仕様はクラス仕様の javadoc をそのまま使える。しかも ssh や https より堅牢なユーザ認証や暗号化は RCI では最初から用意されている。

■RCI が目指すもの

インターネットの世界はほとんど無法地帯と化しており、その構造から巧妙な攻撃者を特定することはとても困難であり、ほとんど反撃できないため、徹底して守りを固めるのがほとんど唯一の対策になる。暗号化されている https や ssh も、ユーザ認証において総当たり攻撃（ブルートフォースアタック）を受けると突破される脆弱性がある。安全だと信じられている暗号化自体も実は疑わしい。このような脆弱性を回避し、開かれたネットワーク経由で安全に情報交換できなければ、インターネットの存亡も怪しくなる。RCI はよりセキュアな通信を行える基盤となることを目指している。

開かれたネットワークに置かれる Web サーバはあらゆる攻撃を受ける可能性があり、攻撃者に侵入される可能性は誰も否定できない。（セキュリティパッチが当てられるということはセキュリティホールがあったということだ。）万一、攻撃者に侵入され、SQL で DB サーバから個人情報等の機密情報を盗まれるようでは企業の存続にも関わることになる。

このような場合に Web サーバと DB サーバの中間に DBA(DB Access) サーバを置き、Web サーバと DBA サーバを RCI で中継していたなら、情報漏洩は避けられる可能性が高い。通常 DBA サーバのような中継サーバを置くと開発工数が大きくなるが、RCI ならば、プログラムの分散配置が容易になる。このような防衛方法は、戦国時代の城のように本丸の外に二の丸、三の丸で防衛する方法と似ている。たとえ Web サーバに侵入されても本丸の DB サーバまでは容易に到達できない。RCI は簡単にクラスを遠隔実行できる仕組みを提供することだけではなく、中継サーバを設けるセキュリティ対策にも使える。

IoT で様々なものがネットに接続されるようになると言われている。それには十分なセキュリティ対策をする必要があるが、RCI を使えば簡単にできる。

java は様々なマシン上で動くということが良さであるから、RCI は PC やスマートフォン、タブレット、各種マシン間のセキュアな通信に利用できる。

RCI はプログラムを実行するマシンを容易に分散できるので、多数のコンピュータを並列運用する負荷分散や科学技術計算にも利用できる。

RCI は java クラスをリモートサーバで実行できるのでクライアントのアプリケーション構造はローカルで単独に実行するように自由度が高い。プログラムはクラスを使って構成するが、そのクラスがどこにあろうと同じようにクラスを使えればプログラム構造は変わらない。

要約すれば、以下の 4 点で RCI は有効である。

・中継サーバを置くセキュリティ対策
・ネットワークの両端のマシン間のセキュアな通信
・負荷分散や科学技術計算
・自由度が高いリッチなアプリケーション開発

■RCI の特徴

リモートの java クラスをローカルで実行しているような簡便さで実行できる。

実行可能なクラスは全クラス形式（インスタンス、スタティック、シングルトン)。

電文(パケット)は人間にも理解しやすい xml 形式（暗号化時は Base64)。

サーバ側で起動した swing(GUI)のキーイベントのような非同期通知もクライアントで受け取ることができる。

パスワードの複数回のハッシュ暗号化を行い、ssh, https 以上にセキュアなユーザ認証を行える。ユーザ認証するならコネクションを維持するのが正当な方法である。

複数回暗号化した電文により格段にセキュアな通信を行える。

階層構造のユーザ管理を行い、各ユーザの権限を階層的に細かく設定できる。

java はプラットフォームを選ばず、どのような OS でも実行可能なのが特徴であり、unix, windows, android, ios 等の異なる OS 間のクライアントとサーバで情報交換できる。

RCI は TCP のコネクションを維持する通信形式のため、http のように毎回接続する性能的に効率が悪いクライアントプルだけではなく、サーバプッシュ型の通信も行える。サーバからの通知は負荷が大きいポーリングではなく負荷が小さいイベント通知でリアルタイムに処理できる。RCI には http のような機能制限がなく、クライアントとサーバ間の相互通信が可能なため機能的にアプリケーションは特にリモートのサーバを利用していることを意識しない構造にできる。http にはこのような制限があるため RCI のパケットを http に載せる RCI on http のようなことは敢えてやらない。コネクションが切れるとサーバで維持していたクラスのインスタンスの参照が失われるのでガベージコレクタの対象になる。この点でもコネクションを維持する方式がよい。

java web start や applet のような仕組みを使えば、Web ブラウザから起動し、RCI 通信でその後の処理を行える。GUI によるリッチなクライアントにすることができ、Web システムと同じようにアプリケーションを事前にクライアントへインストールせずに済む。

http は汎用性がある Web ブラウザが理解できるように公開された情報をクライアントに送っており、サーバとの通信形式を第三者が容易に調べることができる。第三者がその通信形式を模して Web サーバを攻撃することができる。第三者がオリジナルの Web サイトそっくりのなりすましのサイトを容易に作成することができる。この点で html による普通の Web システムは安全なものではない。元々静的なページの不特定多数への公開用のプロトコルであった http が銀行の現金の振り込みに使われるとは当時の設計者にとっては想定外である。java web start や applet で起動した java アプリケーションは独自仕様になるので、内部でどのような通信をサーバと行っているかを第三者が調べることは容易ではない。なりすましのサイトを第三者が作成することも困難である。この点で tomcat 等を使った Web ベースのシステムよりセキュアなシステムをユーザに提供することができる。

■RPC 比較表

基本的に RCI は既存の各種 RPC の欠点を克服することを目的に開発されている。

№	*比較項目*	*RCI*	*ssh*	*https*	*SOAP*	*CORBA*
1	クライアントプル	○	○	○	○	○
2	サーバプッシュ（非同期通知）	○	○	×	×	×
3	ユーザ認証	○	○	△(*1)	△(*2)	△(*4)
4	複数回ハッシュ暗号ユーザ認証	○	×	×	×	×
5	階層構造のユーザ権限設定	○	×	×	×	×
6	パケット暗号化	◎	○	○	△(*3)	△(*4)
7	パケット形式	xml	―	html	xml	―
8	アプリケーション開発容易性(*7)	○	△	△	△	△
9	リッチクライアント	○	×	×	×	×
10	アプリケーション分散配置(*8)	○	×	×	×	×
11	異機種間接続性	○(*6)	×	×	×	×
12	オープンソース、フリーソフト	○	○	○	○	△(*4)
13	設置容易性	○	○	×(*5)	×	×
14	ソケットコネクション維持	○	○	×	×	△(*9)

*1:セッション ID を用いてアプリケーションレベルでユーザ認証を行う。

*2:ws-security というハッシュ化されたパスワードを認証に用いる仕組みはある。

*3: SOAP over https にて実装する。

*4:ベンダー依存。

*5:java を動かす環境構築は簡単ではない。

*6:RCI は単独でサーバとクライアント両方の機能を持ち機種を選ばない。

*7:ssh で繋いでインタプリタ系の言語を使えば遠隔でプログラムを実行するのは簡単にできるが、ssh が扱うのは sh(シェル)で本格的なシステムのインタフェースとしては(winscp のような立派なものも存在するが)あまり利用されない。ssh が sh を遠隔実行するのに対し、RCI は java のクラスを遠隔実行する。java で書かれたシステムであれば、シームレスなインタフェースになる。

*8:分散配置した方が、負荷分散、複雑性の軽減、高可用性、拡張性の点で有利。ソケット通信は単なるプロセス間通信なので同一マシン上で動いていたものを異なるマシン上で動かす変更が容易にできる。同一マシン上で動く RCI クライアントと RCI サーバの数は（RCI サーバのポート番号を変えれば）特に制限がない。

*9:クライアント、サーバが同一マシン上にあり、サーバ同士が通信する複雑なもの。

java の全てのメソッドの形式を含む統一理論のような形式は以下である。

```
Object methodName(Object[] args) throws Exception
```

厳密には Exception ではなく Throwable にすべきところもあるが、Throwable のもう 1 つのサブクラスの Error はほとんど使われることがないので除外し、Exception にするのが妥当である。

java はリフレクションの機能により、クラス名、メソッド名からプログラムを実行できる。上記形式でラップすれば、原則的に全てのクラスの全てのメソッドをリモートサーバで実行し、結果をクライアントで受け取ることができる。

Exception もクライアントで受け取れる。昔はどこで問題が発生したかを知ること自体がかなり困難であったが、Exception には printStackTrace で例外が発生した箇所がすぐわかるという java の優れた特性がある。クライアントにサーバ側の例外発生箇所の詳細を知らせるのはセキュリティ的に問題があるという考えもあるが、プログラムを分散実行しており、クライアントではサーバで実行しているものもクライアントで実行しているかのように扱うには、サーバの例外発生箇所をクライアントで知ることができるのは問題解析のためには都合がよい。Exception クラスは Serializable であるから、オブジェクトを転送しクライアントでサーバの例外発生箇所を知ることもできる。

パラメータ領域(args)で与えられたクラスがサーバ側で変更された場合にも、シリアライズ可能なクラスであれば、クライアント側に反映される。

■アクセス制限

特定機能を特定ユーザのみに実行可能にするために以下のアクセス制限を行っている。

・クライアント IP アドレス（IPv4, IPv6）による接続制限
・ユーザ名、ハッシュ化暗号パスワード認証による接続制限
・ユーザ毎の、パッケージ名、クラス名、メソッド名の実行制限

基本的にアクセス制限をしなければどんなことでも実行できてしまう危険なものであるのは ssh と同じであるから、適切なアクセス制限をしなければならない。

java.lang.System.exit を実行すると RCI サーバプロセスは終了してしまうが、ssh でシャットダウンコマンド（shutdown –h now）を許可しているように、そのような用途も無いとは言えないので、特にシステム共通として制限していない。ユーザ毎に制限することはできる（後述するパッケージ名の制限により java.lang パッケージの実行を禁止することができる）ので、そのようにするのが原則である。

■クライアント IP アドレスのアクセス制限

クライアントとして接続可能な IP アドレスと不可能な IP アドレスを設定ファイルで登録する。

クライアント IP アドレスに応じて、パケット暗号化の要否、ユーザ認証の要否を設定できる。イントラネット内の運用ではパケットを監視するためや性能向上のため暗号化されていない方がよい場合もある。(ユーザ認証しない場合は"anonymous"ユーザになる。)

No.	*暗号化 (encryptPacket)*	*ユーザ認証(userCertification*	*アクセス制御*
1	true(○)	true(○)	暗号化, ユーザ認証
2	true(○)	false(×)	暗号化、非ユーザ認証
3	false(×)	true(○)	非暗号化、ユーザ認証
4	false(×)	false(×)	非暗号化、非ユーザ認証

拒否(ipAddressDenyFilter)と許可(ipAddressAllowFilter)の HashMap の key に IP アドレスを保持し管理している。

原則的に未登録なら許可し、登録されていれば適合した場合のみ許可する。拒否と許可では拒否を優先する。

具体的には以下のパターンがある。

No.	*拒否 (ipAddressDenyFilter)*	*許可(ipAddressAllowFilter)*	*アクセス制御*
1	未登録	未登録	全許可(○)
2	未登録	適合	許可(○)
3	不適合	未登録	許可(○)
4	不適合	適合	許可(○)
5	未登録	不適合	拒否(×)
6	不適合	不適合	拒否(×)
7	適合	―	拒否(×)

サブネットのような IP アドレスのドット(.)区切りの分割登録が可能である。"192.168.3.1"なら、"192.168.3"で登録されていれば適合する。("192.168.3.1/24"のような形式の方が一般的であるが、HashMap のキーで検索するには向かないため。)

■ユーザ管理

RCI サーバにログインし利用できるクライアントユーザの管理は階層構造を採用している。世間一般の組織の人員管理が階層構造を用いているのと同じである。

とても古い OS である unix のユーザ管理は、root, group, user の３段階しかなく階層構造ではない。ユーザ管理を階層構造にすれば、実世界をモデル化した管理がコンピュータシステムでも可能になる。

具体的なユーザ管理の設定方法の概要は以下である。

・ユーザ管理情報は階層構造であるディレクトリの設定ファイルに記述する。
・各ディレクトリには１つのグループ管理情報に相当する設定ファイルを置ける。
・グループ管理情報は階層構造に従い、上位のグループ管理情報を継承する。
・ユーザ管理情報はグループ管理情報を継承する。

１つの部署に相当するディレクトリに従属するユーザに共通する実行権限の管理情報はグループ情報の設定ファイルに記述しておけば、個別ユーザ毎に重複して管理情報を記述する必要がない。

■ユーザ(グループ)管理情報

ユーザ管理情報には以下の項目がある。グループ管理情報はユーザ管理情報を継承しており、同じ項目を持っているが、グループ管理情報のユーザ名は"/"区切りのディレクトリ名になる。

№	*項目名*	*変数名*	*概要*
1	ユーザ名	name	ログインユーザ名を保持する。
2	パスワード	password	ログイン時のハッシュで暗号化されたパスワードを保持する。
3	グループ名	group	ユーザが属するグループ名(/区切りの階層ディレクトリ名)を保持する。
4	ハッシュアルゴリズム名	hashAlgorithmName	パスワードを暗号化する際のハッシュアルゴリズム名。以下のいずれかを設定する。 MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512
5	登録ハッシュ回数	passwordRegistHashRoundCount	設定ファイルに登録するパスワードのハッシュを繰り返した数。

			例えば、2 ならば 2 回ハッシュ化を繰り返したものが設定されるので元のパスワードはわからない。
6	認証ハッシュ回数	passwordAuthHashRoundCount	ユーザ認証時に送られてくるパスワードのハッシュを繰り返した数。サーバで受け取るパスワードがハッシュで暗号化されているのでログを仕込んでも元のパスワードはわからない。 (passwordRegistHashRoundCount 以上であること) (ssh のパスワード認証は暗号化された電文中に平文のパスワードが載っている。ログを仕込めば元のパスワードはわかる。)
7	公開鍵	publicKey	ログイン認証時に ssh や https のように秘密鍵と公開鍵のペアの暗号アルゴリズムを利用しているが、公開鍵を設定ファイルに登録しておくことで、パスワードの認証に加え、公開鍵の一致判定も行う。(セキュリティは更に強力になる。)
8	拒否パッケージ名	packageDenyFilter	アクセス不可にする java のパッケージ名の一覧(例、java.util.HashMap なら、java.util の部分)
9	許可パッケージ名	packageAllowFilter	アクセス許可する java のパッケージ名の一覧 （詳細ロジックは後述）
10	クラス名、メソッド名制限	classNameFilter	クラス名とメソッド名のアクセス制限の定義の一覧 （詳細ロジックは後述）
11	子ユーザ一覧	childFilter	被管理対象のユーザ名をここに登録することで、sudo のようにそのユーザ名としてログイン可能にするためのもの。(実装予定)

■パッケージ名のアクセス制限方式

パッケージ名(例、java.util.HashMap なら java.util の部分)でアクセス制限を設定することができる。

拒否(packageDenyFilter)と許可(classNameFilter)の HashMap の key にパッケージ名を保持し管理している。(value は現状未使用のコメント相当)

原則的に未登録なら許可し、登録されていれば適合した場合のみ許可する。拒否と許可では拒否を優先する。

具体的には以下のパターンがある。

No.	*拒否 (packageDenyFilter)*	*許可(classNameFilter)*	*アクセス制御*
1	未登録	未登録	全許可(○)
2	未登録	適合	許可(○)
3	不適合	未登録	許可(○)
4	不適合	適合	許可(○)
5	未登録	不適合	拒否(×)
6	不適合	不適合	拒否(×)
7	適合	—	拒否(×)

■クラス名、メソッド名のアクセス制限方式

クラスの完全修飾名、メソッド名でアクセス制限を設定することができる。

上位グループの定義を継承する。

原則的にクラスが未登録なら全て許可し、登録されていれば適合したクラスのみ許可する。メソッド名も未登録ならそのクラスのすべてのメソッドを許可し、登録されていれば適合したメソッドのみ許可する。

具体的には以下のパターンがある。

No.	*クラス名*	*メソッド名*	*アクセス制御*
1	未登録	—	全許可(○)
2	適合	未登録	許可(○)
3	適合	適合	許可(○)
4	適合	不適合	拒否(×)
5	不適合	—	拒否(×)

■複数回ハッシュ暗号化パスワード認証方式

ハッシュ関数は片方向の暗号化を行うものでハッシュ化されたパスワードを用いてユーザ認証を行うことに広く利用されており、unix ユーザのログイン認証もこの方式を採用している。

例えば、”pass1”という文字列を” SHA-256”というハッシュアルゴリズムでハッシュ化し、16 進数文字列とすると

e6c3da5b206634d7f3f3586d747ffdb36b5c675757b380c6a5fe5c570c714349

になる。サーバ側では e6c3…というハッシュ化された文字列を保持しておき、パスワードとして与えられた”pass1”という文字列をハッシュ化し、保持された文字列と一致判定を行うことで与えられたパスワードが正しいかを判断する。このことは同じパスワードを同じアルゴリズムでハッシュ化すると常に同じハッシュ値になる特徴を利用している。

このようなパスワード認証によれば、サーバ管理者すらユーザのパスワードが何かを知る術がないと考えられているが、サーバ側では”pass1”という平文のパスワードを取得しなければ、パスワード認証ができなかったのであるから、何かログを仕込む等の細工をすれば、サーバ側でユーザのパスワードが何かを調べることができる。つまり、サーバ管理者であれば、ユーザのパスワードが何かを知る術はある。本来誰にも知られてはいけないのが原則のパスワードがネットワーク越しにサーバに送られるというのは、ユーザがどんなに努力しようと秘密にできないことになる。秘密というのは誰にも知られないようにすることであるが、方式的にサーバに知らせているのである。なりすましのサイトだと気付かずパスワードを入力して「パスワードを盗まれた」後に正規サイトに転送されるとほとんど不自然さを感じない。

何が問題なのか。平文のパスワードが（パケットは暗号化されていても）サーバに渡されるという方式ではパスワードは秘密にならないということであり、方式的な一種のセキュリティホールである。対策は平文のパスワードをネットワークに流さないことになる。

ハッシュを繰り返した結果も同じハッシュ値になるので、クライアント側で複数回ハッシュ化を繰り返したものをネットワークに流し、サーバ側でもハッシュ化されて保持されている値のハッシュを繰り返せば認証できる。ネットワークに平文のパスワードは流れず、ハッシュ値が流れるので、サーバ側で元の平文のパスワードが何かはわからない。

ハッシュ回数を任意に設定できるようにすれば、ハッシュ値は複雑なものであるから、ユーザがパスワードを簡単なものではなく、推測されにくい辞書に無いような複雑なものにする必要も特にない。システムはなるべくユーザに難しい操作を求めないようにすべきである。

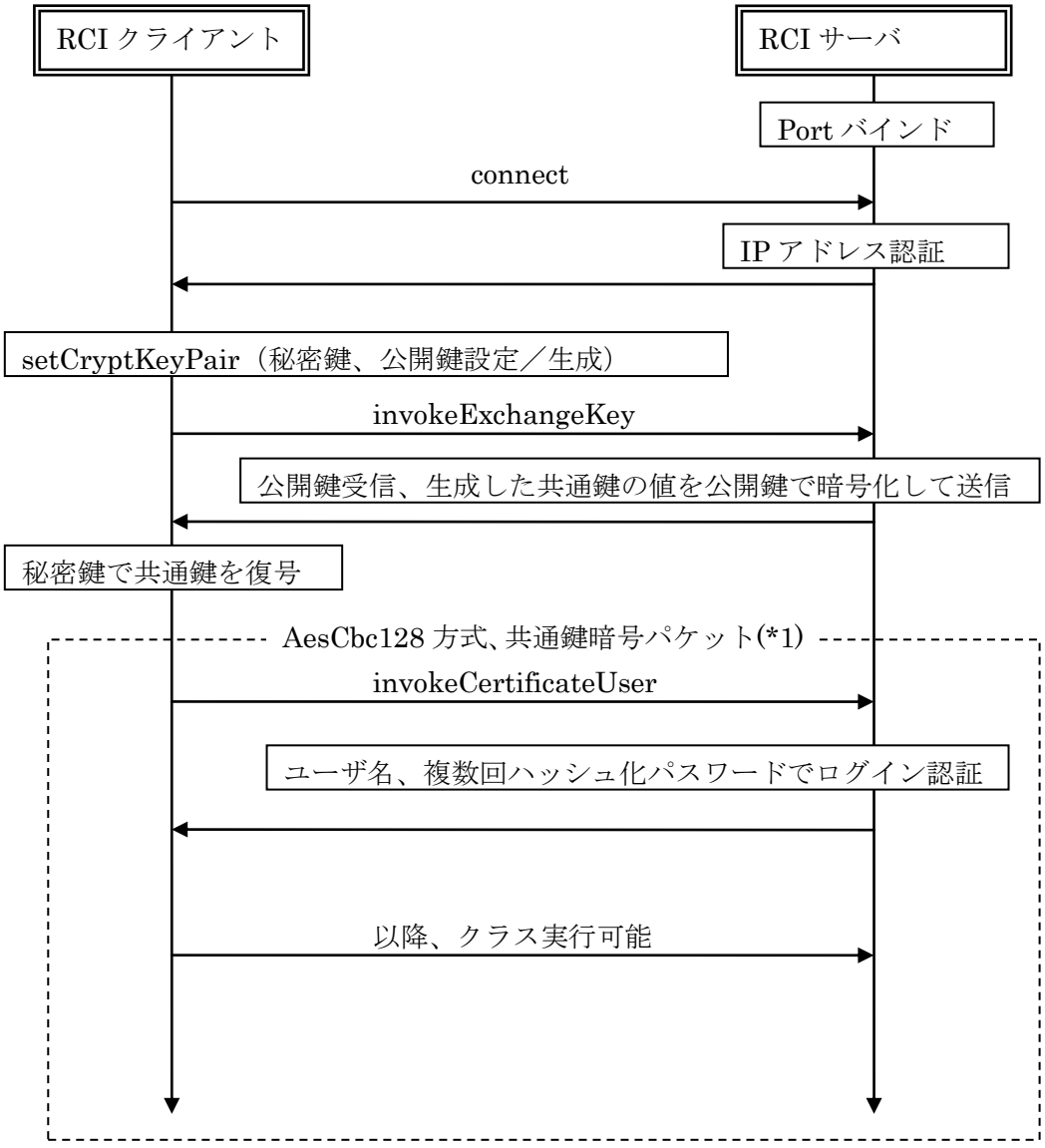
■複数回ハッシュ暗号化パスワード認証方式のパターン

No.	*ログイン用 (クライアント) ハッシュ回数(n)*	*照合用 (サーバ) ハッシュ回数(m)*	*認証方法*	*評価*
1	0(平文)	0(平文)	・n(0)=m(0)の場合 ・平文のパスワード認証 ・原始的な方式	×
2	0(平文)	1	・n(0)<m(1)の場合 ・unix 等、一般的な認証方式 ・平文のパスワードをサーバで 1 回ハッシュ化し照合 ・サーバでは平文のパスワードが必要 ・電文を盗み見られるとログイン可能	△
3	1	1	・n =m の場合 ・送られて来たハッシュをそのまま照合 ・サーバで平文のパスワードは不要 ・電文を盗み見られるとログイン可能	△
4	1	2	・n<m の場合 ・クライアントから送られて来たハッシュ化されているパスワードをもう 1(m-n)回ハッシュ化し照合 ・サーバで平文のパスワードは不要 ・電文を盗み見られてもログイン不可能	○
5	2	1	・n>m の場合 ・サーバで保持されているハッシュ化されているパスワードをもう 1(n-m)回ハッシュ化し照合 ・サーバで平文のパスワードは不要 ・電文を盗み見られてもログイン不可能	○

ログイン用ハッシュ回数と照合用ハッシュ回数はサーバ側で保持し電文には載せない。それぞれの回数の上限を N とすれば、組み合わせ数は N*N 個になる。N が 100 なら、10000 通りのパターンになるので、総当たり攻撃は極めて困難になる。

古典的な１や２のパターンも設定可能な方法に含まれており一般的な認証方法であることがわかる。

■ログイン認証シーケンス



*1:キー長は 256 ビットが望ましいが、java の標準ライブラリの制限から標準では 128 ビットにしている。256 ビットに変更するには、${JAVA_HOME}/jre/lib/security の local_policy.jar と US_export_policy.jar を無制限のものに置き換え、com.graveng.rci.crypt.AesCrypt. KeyLength を 16 から 32(256 ビット)に変更する。(不特定多数のクライアントを想定する場合、この変更は難しい。キー長が 128 ビットでも暗号強度を格段に高める方法については次項にて述べる。)

■暗号解読の方法と対策

Aes-Cbc-128（Advanced Encryption Standard, Cipher Block Chaining Mode）の共通鍵暗号方式は128ビットの共通鍵と128ビットの初期ベクターを鍵として暗号化している。

8ビットの数値空間は2の8乗で、0～255(2^8)の値を持てるように、Aes-Cbc-128の鍵は初期ベクターを含めて128+128ビットなので、2の256乗で0～2^256の数値範囲になる。鍵は0～2^256のどれかになる。

この数値を0からインクリメント(+1)し、単純に復号化を繰り返せばいつか鍵に当たる。復号化した際に解読できたと判断するには、テキストになったとか、数値配列のようなパターンが現れたとかの方法があるだろう。

この暗号解読の計算時間が長いだろうから、暗号解読は難しいとされているが、この計算は並列処理できる。多数のマシンに鍵の数値範囲を分担し並列で計算させ、マシンの数を増やせば、理論上、短時間で暗号解読できる。計算時間が最大100日なら、100台で分担すれば1日未満で終わる。Aes暗号は16バイトのブロック毎に暗号化している。暗号化されているファイルの暗号化前がテキストだとわかっているなら、最初の16バイトだけを復号してみれば、結果がテキストかそうでないかは直ちにわかる。テキストでなければその鍵は正しくないので、次の鍵を調べる。このようにすれば暗号解読の計算時間は劇的に短縮できる。httpsはhtmlのテキストを転送しており、独立したWebブラウザとWebサーバ間の通信なので暗号化アルゴリズムは公開されており、おそらく大規模機関にとって解読は何も難しくないだろう。

Aes-Cbc-256で共通鍵を256ビットにして強度を上げてみたところで、短時間で暗号解読できる方法があることは変わらない。

問題は復号化した際に暗号解読できたと判断する方法があることである。もし、復号化した結果に何も特徴がなければ、復号に成功したと判断することができないので、何が鍵だったのかわからず暗号解読はできない。(java標準のjavax.crypto.Cipherクラスで"AES/CBC/PKCS5Padding"のパディング設定で暗号化した場合、共通鍵や初期ベクターが異なるもので復号化しようとするとjavax.crypto.BadPaddingException: Given final block not properly padded例外になり、ご丁寧に共通鍵や初期ベクターが正しくないことを教えてくれる。これは便利な面もあるが、暗号解読が容易になる脆弱性にもなる。パディングしている領域に何か共通鍵や初期ベクターが正しいか検証できる拡張的な情報（8ビット以下のチェックサムのようなものをストッパーにしているのだろうか）が含まれていることになる。この対策は"AES/CBC/NOPADDING”の設定とし16バイトの倍数にならない場合は暗号化前のデータが16の倍数になるようにxmlならばSPACE埋めのようなことをすればよい。)

この対策として、パスワードのハッシュ化を繰り返したように、Aes-Cbc-128のパケットの暗号化／復号化も繰り返せばよい。

A) 一般的な１回の暗号化／復号化

[暗号化] テキスト → 暗号化 → バイナリ暗号文

[復号化] バイナリ暗号文 → 復号化 → テキスト

B) 複数回繰り返し暗号化／復号化

[暗号化] テキスト → 暗号化１ → バイナリ暗号文１ → 暗号化２ → バイナリ暗号文２

[復号化] バイナリ暗号文２ → 復号化２ → バイナリ暗号文１ → 復号化１ → テキスト

※バイナリ暗号文 2 を復号化してバイナリ暗号文 1 を得ても解読できたという特徴が検出できないので鍵はわからないため、その鍵で更に復号化しテキストを得ることはできない。繰り返し回数がわからないか、２回目以降の暗号化／復号化の鍵が異なっていれば単純に２回繰り返しても解読できない。

より堅牢な暗号化方式としては以下のような実装が望ましい。

(1) 鍵検証を避けるために javax.crypto.Cipher は"AES/CBC/NOPADDING”の設定とし、パディングは独自に行う。

(2) 暗号化／復号化は複数回連続して行えるようにする。連続して行う方式は時系列が現れるので CBC のように２回目以降を分割して並列処理することができない。単に力技で多量のマシンを投入すれば短時間で解読できるものではない。繰り返しは単純な方式であるが、暗号生成のコストが回数倍になるのに対し、その解読コストは回数乗倍になる。

(3) 複数の暗号化／復号化の共通鍵と初期ベクターはそれぞれ別に生成したものを使う。

(4) 暗号化／復号化の繰り返しはバイナリからバイナリとし解読できた特徴を無くす。

RCI では javax.crypto.Cipher は"AES/CBC/PKCS5Padding"の良さもある(パディングをアプリケーションでやらずに済む、不適切な共通鍵や初期ベクターを検出できる)ので、(1)は変更できるようにしている。デフォルトで" AES/CBC/NOPADDING "としているが、パケットは xml のテキストなので SPACE で一律パディング可能である。以下のように設定すれば変更もできる。

com.graveng.rci.crypt.AesCrypt.Algorithm = " AES/CBC/PKCS5Padding";

(2)、(3)、(4)は後述する ServerConfig.xml の設定で変更できるようにしている。(以下はパケットの暗号化／復号化回数を 3 にする例。デフォルトは 2。この数字を大きくすると応答速度は劣化する。1 つめを解読する方法がないので、3 でも十分固い。総当たりのパターン数は(2^256) ^3 にもなる。)

```
<void property="aesCryptCount">
 <int>3</int>
```

</void>

https や ssh の暗号化／復号化回数は一律１回である。将来コンピュータの性能が上がると既存の暗号は無意味になると危惧されている。１回では安心できないが、任意の設定回数分暗号化が繰り返されており、暗号解読できた特徴も検出できないなら、暗号解読しようとする気力を失うだろう。

■設定ファイル一覧

user.dir が rci_server ならば、各種ファイルは rci_server 配下に配置する。

```
rci_server （設定ファイル配置例）
├──data // rci に関するデータ
│    └──Cm  // 設定ファイルに関するもの
│          │   ServerConfig.xml  // RCI サーバの設定ファイル
│          │
│          ├──hosts  // RCI クライアントの接続認証に関する設定ファイル
│          │    ├──allow
│          │    │        .xml  // デフォルトの設定ファイル
│          │    │        127.0.0.1.xml  // IP アドレスが 127.0.0.1 に関する設定
│          │    │
│          │    └──deny
│          │             0,0,0,0,0,0,0,1.xml  // IPv6 の 0:0:0:0:0:0:0:1 の場合(:は,で判定)
│          │             192.168.3.11.xml
│          │             192.168.2.xml // 192.168.2.*を拒否する場合
│          │
│          └──user  // RCI クライアントユーザのアクセス権限に関する設定ファイル
│               └──第 1 事業部
│                    └──第 1 システム部
│                         │   .xml  // グループ（第 1 システム部）の設定ファイル
│                         │
│                         └──RCI 開発課
│                                  .xml  // グループ（RCI 開発課）の設定ファイル
│                                  山田太郎.xml  // ユーザ（山田太郎）の設定ファイル
│                                  山田次郎.xml
├──log
│        logfile.txt  // ログファイル
```

■設定ファイルの記述方法

各種 xml の設定ファイルはオブジェクトを java.beans.XMLEncoder でファイルに保存し、ファイルからは java.beans.XMLDecoder でオブジェクトにロードしている。

(1) ServerConfig.xml　// RCI サーバの設定ファイル例

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_92" class="java.beans.XMLDecoder">
 <object class="com.graveng.rci.server.config.ServerConfig">
  <void property="logLevel">
   <int>2</int><!--①-->
  </void>
  <void property="logFileName">
   <string>logfile1.txt</string>
  </void>
  <void property="port">
   <int>5243</int><!--②-->
  </void>
  <void property="serverCertification">
   <boolean>true</boolean><!--③-->
  </void>
  <void property="password"><!--④-->
   <string>e6c3da5b206634d7f3f3586d747ffdb36b5c675757b380c6a5fe5c570c714349</string>
  </void>
  <void property="hashAlgorithmName">
   <string>SHA-256</string>
  </void>
  <void property="passwordAuthHashRoundCount">
   <int>0</int>
  </void>
  <void property="passwordRegistHashRoundCount">
   <int>1</int>
  </void>
  <void property="wrappedException">
   <boolean>true</boolean><!--⑤-->
  </void>
```

```
  <void property="initializeClass">
   <string>example.test.RciServerInitializeImpl</string><!--⑥-->
  </void>
 </object>
</java>
```

① ：logLevel； VERBOSE(2),DEBUG(3), INFO(4), WARN(5), ERROR(6)を設定可能。デフォルトは INFO(4)

② ：port； RCI サーバが待ち受けるポート番号。(RC のアスキーコードから勝手に採番)

③ ：serverCertification；RCI サーバ(RciServer)起動時にパスワード認証を行う場合は true にする。デフォルトは true

④ ：password , hashAlgorithmName 等は RCI サーバ起動時のパスワード認証用

⑤ ：wrappedException；サーバで発生した例外をシリアライズしてクライアントに転送する場合は true、発生した例外のクラス名とメッセージを RciException に載せ換えてクライアントに転送する場合は false を設定する。デフォルトは true。

⑥ ：initializeClass: RCI サーバ起動直後に何かユーザ独自の初期化を行いたい場合に、ここに登録されているクラスが実行される。

(2) hosts/allow/127.0.0.1.xml　// RCI クライアントの接続認証に関する設定ファイル例

ファイル名はクライアントの IP アドレス。

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_92" class="java.beans.XMLDecoder">
 <object class="com.graveng.rci.server.config.IpAddressConfig">
  <void property="encryptPacket">
   <boolean>true</boolean><!--①-->
  </void>
  <void property="userCertification">
   <boolean>true</boolean><!--②-->
  </void>
 </object>
</java>
```

① :encryptPacket パケットを暗号化する場合は true、暗号化しない場合は false にする。ユーザ認証を行う場合はパケットを暗号化する。

② ：userCetification ユーザ名とパスワード認証によるログインを行う場合は true にする。パスワード認証を行わない場合のユーザ名は”anonymous”になる。anonymous の設定ファイルは user/anonymous.xml になる。

deny の下のファイルはファイル名のみ意味を持ち、ファイルの内容は無効で以下のようする。

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_92" class="java.beans.XMLDecoder">
 <object class="com.graveng.rci.server.config.IpAddressConfig">
 </object>
</java>
```

(3) 山田太郎.xml　// ユーザ（山田太郎）の設定ファイル例

以下の xml 形式には、void method や object class があり、フィルター対象と誤認しそうになるが、java.beans.XMLEncoder が認識するクラス名とメソッド名のことである。xml の構造を個別に定義するのはコストが高く、XMLDecoder, XMLEncoder を使えばそのようなコストは避けられる。

グループの設定ファイルも形式は同じでファイル名が".xml"になる

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_92" class="java.beans.XMLDecoder">
 <object class="com.graveng.rci.server.config.UserConfig" id="UserConfig0">
   <void property="packageDenyFilter">
   <void method="put">
    <string>java.util</string><!--①-->
    <string>実行不可のパッケージ名を指定している(コメント)</string>
   </void>
  </void>
  <void property="classNameFilter">
   <void method="put">
    <string>java.lang.System</string><!--②-->
    <object class="java.util.HashMap">
     <void method="put">
      <string>getProperty</string><!--③-->
      <string>true</string>
     </void>
    </object>
   </void>
   <void method="put">
    <string>java.util.Date</string><!--②-->
    <object class="java.util.HashMap"/>
   </void>
  </void>
    <void property="group">
  <string>第 1 事業部/第 1 システム部/RCI 開発課</string><!--④-->
  </void>
  <void property="hashAlgorithmName">
```

```
    <string>SHA-512</string>
   </void>
   <void property="name">
    <string>山田太郎</string><!--⑤-->
   </void>
   <void property="password">
    <string>pass1</string><!--⑥-->
   </void>
   <void property="passwordAuthHashRoundCount">
    <int>0</int>
   </void>
   <void property="passwordRegistHashRoundCount">
    <int>0</int>
   </void>
  </object>
</java>
```

① ：packageDenyFilter；実行不可のパッケージ名の指定
② ：実行を許可するクラス名
③ ：実行を許可するメソッド名
④ ：所属するグループ名は省略可（ディレクトリ名から内部変数に設定されるのでファイルに書く必要はないが、プログラムでファイルにセーブする際には書かれる。）
⑤ ：ユーザ名は省略可（ファイル名から内部変数に設定）
⑥：password, hashAlgorithmName, passwordAuthHashRoundCount, passwordRegistHashRoundCount はユーザ認証用のパスワードに関する記述。この例はハッシュで暗号化していない不適切なもの。

・java.lang.System.getProperty(“user.dir”)を遠隔実行する要求電文例

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_92" class="java.beans.XMLDecoder">
 <object class="com.graveng.rci.Packet" id="Packet0">
  <void property="optionMap">
   <void method="put">
    <string>paramTypes</string>
    <array class="java.lang.Class" length="1">
     <void index="0">
      <class>java.lang.String</class>
     </void>
    </array>
   </void>
  </void>
  <void property="params">
   <array class="java.lang.Object" length="4">
    <void index="0">
     <string>static</string><!—static, new, single の形式指定-->
    </void>
    <void index="1">
     <string>java.lang.System</string><!—クラス名-->
    </void>
    <void index="2">
     <string>getProperty</string><!—メソッド名-->
    </void>
    <void index="3">
     <string>user.dir</string><!—パラメータ-->
    </void>
   </array>
  </void>
  <void property="requestId">
   <long>1522368251957</long>
  </void>
  <void property="serializedParams"><!—パラメータ領域のクラスをシリアライズして転送する領域-->
```

```
    <array class="java.lang.String" length="4"/>
   </void>
  </object>
</java>
```

・java.lang.System.getProperty(“user.dir”)を遠隔実行する応答電文例

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_92" class="java.beans.XMLDecoder">
 <object class="com.graveng.rci.Packet" id="Packet0">
  <void property="optionMap">
   <void method="put">
    <string>paramTypes</string>
    <array class="java.lang.Class" length="1">
     <void index="0">
      <class>java.lang.String</class>
     </void>
    </array>
   </void>
  </void>
  <void property="params">
   <array class="java.lang.Object" length="4">
    <void index="0">
     <string>static</string>
    </void>
    <void index="1">
     <string>java.lang.System</string>
    </void>
    <void index="2">
     <string>getProperty</string>
    </void>
    <void index="3">
     <string>user.dir</string>
    </void>
   </array>
  </void>
  <void property="requestId">
```

```
    <long>1522368251957</long>
   </void>
   <void property="result">
    <string>C:¥pleiades¥workspace¥rci</string>
   </void>
   <void property="serializedParams">
    <array class="java.lang.String" length="4"/>
   </void>
  </object>
</java>
```

■依存ライブラリ

(1) ファイル IO 用

[提供元 WebSite]

https://commons.apache.org

[jar]

org.apache.commons.io_2.2.0.v201405211200.jar

[class]

org.apache.commons.io.FileUtils

org.apache.commons.io.filefilter.FileFilterUtils

(2) 暗号化時のランダム文字列生成用

[提供元 WebSite]

https://commons.apache.org

[jar]

org.apache.commons.lang3_3.1.0.v201403281430.jar

[class]

org.apache.commons.lang3.RandomStringUtils

(3) RSAKeyPair の作成用

[提供元 WebSite]

https://www.bouncycastle.org/latest_releases.html

[jar]

bcprov-jdk15on-159.jar

[class]

org.bouncycastle.jce.provider.BouncyCastleProvider

(4) JDK は 1.8

■主なパッケージ構成

No.	パッケージ名	概要	A	J	C	S	E
1	android.util	Android 環境の Log クラスを模したもので、非 Android 環境で Android 用の Log 出力を行えるようにしたもの。Android 環境で動かす場合は不要。(*1)	×	○	—	—	—
2	com.graveng.rci.client	RCI クライアント用のもので RCI サーバとして動かす場合は不要。	—	—	○	×	○
3	com.graveng.rci.server	RCI サーバ用のもので RCI クライアントとして動かす場合は不要。	—	—	×	○	○
4	com.graveng.rci.crypt	暗号化に関するもの	—	—	○	○	○
5	com.graveng.rci.util	各種ユーティリティ	—	—	○	○	○
6	example	主にクライアントのサンプルだが、サーバで動くクラスも含んでいる。（サーバで遠隔実行するクラスはサーバ側に必要。）	—	—	—	—	○

A:Android 環境, J:通常 java 環境, C:Client, S:Server, E:Example
*1:この構想は複雑化させているのでやめた方がいいかも知れない。

■コンパイル方法

build.xml の makejar を ant で実行すると target/lib ディレクトリの下に com.graveng.rci_1.0.0_v201803301530.jar（例）が出来、上記依存ライブラリは target/lib の下にコピーして置かれる。(依存ライブラリの最新版は各提供元から取得する。)

com.graveng.rci_1.0.0_v201803301530.jar には RCI クライアントと RCI サーバの両方のクラス等、上記の全パッケージが収容されている。（不要なパッケージを省きたい場合はそのパッケージを削除してから、makejar を実行する。）

target ディレクトリにはサンプルの設定ファイル群の data ディレクトリもコピーされる。

■環境変数の設定

各種設定ファイルは System.getProperty("user.dir")で取得できるディレクトリ（カレントディレクトリ）からの相対パスで取得している。

起動時にこの環境変数を変更するには以下のようにする。

```
java -Duser.dir=/usr/local/rci -cp lib/* com.graveng.rci.server.RciServer -s 起動パスワード
```

■RCI サーバプロセスの起動と終了

target ディレクトリにカレントディレクトリを移してから実行する。

(1) Windows の場合の起動例

```
java -Dfile.encoding=UTF8 –cp　.¥lib¥* com.graveng.rci.server.RciServer -s pass1
```

(2) Unix の場合の起動例

```
java -Dfile.encoding=UTF8 -cp ./lib/* com.graveng.rci.server.RciServer -s pass1
```

※file.encoding がないと以下のようなエラーになる場合がある。

「com.sun.org.apache.xerces.internal.impl.io.MalformedByteSequenceException: 3 バイトの UTF-8 シーケンスのバイト 3 が無効です。」

※pass1 はサンプルの設定ファイルのパスワード(ハッシュで暗号化したものをパラメータに与えるのが望ましいが、例として簡単にしている)

終了は Unix なら、SIGTERM シグナルを kill コマンドでプロセス ID に向けて送信する。(TAGET/data/pid.txt にプロセス ID はある。)

Windows の場合は Ctrl-C で止まる。

強制終了してもソケットやファイルはクローズされ、メモリも解放されるので、特に問題ない。(安全に停止するためには、新たな要求を受け付けないようにして、要求中の処理の終了を待ってから停止するのが良いが、要求中の処理に問題があって終了させたい場合は困ることになる。回線はいつ切れるかわからないのだから、急に切断されても困らないようにクライアントではなっていなければならない。それはサーバが強制切断するのと同じことであるから、サーバは強制切断してもよい。非定常系の処理に凝ると切りがなく、開発工数を肥大化させるが、それは肥えたシステムのようになる。)

■RCI クライアントサンプルの実行

RCI サーバを起動した target ディレクトリにカレントディレクトリを移してから実行する。

(Windows の場合の起動方法)

```
java -Dfile.encoding=UTF8 -cp . ¥lib¥* example.test.Test0
```

(Unix の場合の起動方法)

```
java -Dfile.encoding=UTF8 -cp ./lib/* example.test.Test0
```

■RCI でのクラス実行と通常のクラス実行の比較

RCI のクラス実行は通常のクラス実行とあまり変わらない簡略さで記述できる。

形式	*RCI のクラス実行*	*通常のクラス実行*
static	Packet result = rci.invoke ("static", "java.lang.System", "getProperty", "user.dir"); System.out.println(result.getResult());	String result = System.getProperty("user.dir"); System.out.println(result);
new	Packet hashMap = rci.invoke("new", "java.util.HashMap");	HashMap hashMap = new HashMap(); // HashMap<String, int> のようなことはプリコンパイルになるので RCI ではできない。
instance	rci.exec(hashMap, "put", "age", 21);	hashMap.put("age", 21);
instance	int number = rci.exec(hashMap, "get", "age").toResult(0);	int number = (int) hashMap.get("age");

■RCI クライアントサンプルプログラム(static, new)

```
public class Test0 {
    public static void main(String args[]) {
        try {
          // RCI サーバのホスト名、ポート番号を指定
            String server = "localhost";
            int port = 5243;

            Rci rci = new Rci();
//                  rci.setUseEventListener(false);
            // ソケット接続
            rci.connect(server, port);
//                  rci.getInvoker().setRecvSleepMs(100);

            System.out.println("before " + new Date());
            System.out.println("鍵交換中");
            rci.invokeExchangeKey();
            System.out.println("after " + new Date());

            System.out.println("ログイン認証");
            rci.invokeCertificateUser("山田太郎", "pass", "SHA-256", 0);

            // static メソッドの実行
            System.out.println("スタティックメソッドの実行");
            Packet result = rci.invoke("static", "java.lang.System", "getProperty","user.dir");
            System.out.println("server の user.dir:"+result.getResult());

            System.out.println("インスタンスメソッドの実行");
            // インスタンス生成
            Packet dateResult = rci.invoke("new", "java.util.Date");

            // 生成したインスタンスのメソッド実行
            dateResult = rci.exec(dateResult, "getMonth");
            System.out.println("今月-1：" + dateResult.getResult());
```

```
            Packet hashMap = rci.invoke("new", "java.util.HashMap");
            rci.exec(hashMap, "put", "age", 21);
            int number = 0;
            number = rci.exec(hashMap, "get", "age").toResult(number);
            System.out.println("server の HashMap に設定した値:"+number);

            // インスタンスの破棄
            rci.exec(dateResult);
            rci.exec(hashMap);

            // ソケット切断
            rci.disconnect();

        }
        catch (Exception e) {
          e.printStackTrace();
          System.exit(3);
        }
    }
}
```

■RCI クライアントサンプルプログラム(single)

```
    public static void main(String args[]) {
        try {
          // RCI サーバのホスト名、ポート番号を指定
            String server = "localhost";
            int port = 5243;

            Rci rci = new Rci();
            // ソケット接続
            rci.connect(server, port);

            // 特定の秘密鍵と公開鍵を使う場合
            String publicKey =
"H4sIAAAAAAAAAFvzloG1uIjBLr8oXS8pvzQvuTI5sbgkJ1UvKzkxKzlVr6AovywzJbVIL7G4Mjc3taQ
```

oM1mvqDhRz8k5KNgxoDQpJzPZO7VSVSmLL/pXTgszA5MPA3tufkppTmlxCYOYT1ZiWaJ+bmJJhr
5TZrpnXklqemqRtQ8DXwFYp2tFQX5eal5JIUMdA2NFAdAdIiANeiANeggNPX/my6+0/i3LzMDmycC
RlFniDHRoiScDJ5Dpk5qXXpLhySCclllUXOKXn1eVWpTvVFmS6lea68nAk5NfnlpcEpxa4pQJ1MBW
nJmeV5obzcCZm5iel1lSmpJawsAU7VRRxCAAtjgnMS9dD6gzKbWobc1UWe4pD7qZGBgqCv4jwD8Q
ZmBgYCwtAmld81n8BxtHyAOwMqAow2STd8xOM7gOrw78qrJph8Ztk65tqyv+ac30q9YsEpd7fra2w9
2twHX6tZnVWc7dzybVRxx7+vvs1ekxrIWdSdEl/Nbzl3rzZz6vlzaqqWAOEu3cI1a57cxrH+UFnxYf9+h
V/7F2yy+VryaP/vbLPDS13jV1W+mH3x3BkZXdUT3rgwI+5bvNDT61mXNhfqEJp1HHmbV/9+mdYNi
Z+t7bm1OfrS/k0tXavRfjn55/V2Vdc68ldE2ZWT7HpKtMW4742cywz9yoOGXn1H/7t99jLRTNnNgSlhy
wSdl1fqSPRlXNk2v3/J2L3Le3VbdcDGJfv7DOU0TW4Zdp0cIQbv16Ix5rE07lJ003+aNvnLpYUQyKTW
asIQaSYQeymBmB0V0BAF4Ug9V7AgAA";

String secretKey =
"H4sIAAAAAAAAAKVVezSUaRz+MjVMQhd0IbtESdvMnnGrJslMRmMmxpBcqzE+0zfN9ZtvGJewic
YlupCERZMuTFREpCi6rpJKVEJMUWq66KJc2mHb7bRn/3DOvue857zv8z3P+T3P8/7xKV4BU8QwQ
BTAbGyQQMJnhbOYYoQLYjksJocFYoWwIBQKBmEsUxzO44EIDLGwsJiJJZIYnk50GAplIiAJRqhgO
Ld65y0N8XqsBoCmAbosGCEJwJAQiAWBfAQBDGkcZigTx2MiW3BEiE3hIyAbhAlqphCGeKCzVCjgq
4l0ERANTPo36vENRY+j9B9uHv8oJEFciPW3ZByVwoDj/wmmTkXux9RpNjkZowANGqDJEwRLuBL
xt5F6wr9oP84UquvUHwuLHQuL/R42ZTjvp+OEofkoAE0BtIIgdUESPkIBMOojDeSzkS0UYFYIBIsR
NwE/AoQFxHAEdJPwKMA0riAMFCOeIEKE1AK0GGLzJTx/AMNjsvkQIgkGEUDDn6iOO318MJfJZ
2PVyiAQTlBkzNdO70rWAACp8Ov3NTq2AQCYJIHHpIqB2YNoLa+ucZoaBfbZqFDEA1Priz0+mJ8+a/
nIJqmiWDpqleUWuRiebdJ7c5vMhSx0zryfFckhJT/bG+Nz5enQzZbMgCmixCB/RI+QV0jVg3pjjPBRUh
TDILHGMLyisZ+2IP+d/OraXYsGT5R/Mf9g0z2SZvzEllCdUSF5MyTz9A1P9kspZdDfCci5njfKMAUCk
Q0GL2s8MXIBew2oAl9TqRgcOtXrTsu2882bnjapIghRHTvWK0LtBFp7WzTKG9xWHnCETpmmV2W
M1lZ2TBEZQHt2eLPopxc45/nSLCOilPc73EmwS2VC5I5mhmZpQTRFf/7qL7ZwgZc2LgY/jWCDWaDc/
kDPv+1Gs3T8kSf/Z2NjX7TGSsLcsGvXLnRQoTlrRMqHnbKz8Z6mZU516Dh9y8ZB0+Lk9t+6cDmp7a
4XHvi0eluh+xfa3jauFS9OcFUEtZitfLpMDOpTfOV2WEZAVUNSg/QVqlXOMhb2+ys2BOqgipYH/jw9/
bz8K/lgERx/i5rV7aFf4nDJibY/KTSrzsPuvtcJ96nP5+EZRBVmsMOinNCL1pNtIsavciNeqCaXlrWZ5mj
dJVjNTAkIy7E/vtC98LLD8BdGrMWQ4/DwTNLd8D1t2u8L/RhN1dI4k9GNoSUfs0wNpm7PmPsojGW
RJ7gjonmsMZi2bKnOi9LsuNhsIPPiwgiHI30SlbMutwdlP6LldC5ZOoGWgFjJy91H7Xmd+X2VuoOPhh
efOTSgIX9ftzmfUnNldvoGSUj5qbk6XR+6rZf6LuLgRIvmHC7ckX2lZlfv4BJT+raQqjRn/MLdykILnB9
GJD9EDZ33fJ+qiPv5+mMPWs5ZU1luYNTmtEtPe4SWBuSadPdFpvza4jjER9DTrGmR/fz12j75hExqh
lHSvFyI/SMaJyFFrNhpcU/85QBRBju6Nq3gUq/+nbMF0ZaTz5ROXvbWddcza8zJe5NR8Zqq1xVaEKf
VZ16SiVmhYgFap9x79WWKeKBh/UPrqd3cOSHUFWk2nsr9kuCPTHR9b73V7Sc+p82BSJVed0NUr8
zWfNDPXdysQiYF90zIpK0cbAtbp6xj2qp215SY7KHo9wjcP55hOfmEzQsx83anNkk3g44ujv0bSrYGM
YtnHHW8JjbunWbYXEPmclB65mtHn5UVnWg5XnAwRlDpUfJ4ayWyNSWgP8Wzzy+ub5Xq7SfGqVf
EkuVH4/mPrFyB8ofPczvdY7Mi2l1jSJXHch2mTMjk9WzdN7r49dcMD/e9WUKUOdCSfy/AWz+hU+jx
YX6nCVJqctusM/0q3c/rShPfmii7Dtsv+1j/JcIxTue8zbmBr0fs8K5J915HzrZPmnPbLL9iPufTr25XNqg2

```
prq0ZuYcamyO5DlbmH3ea8Rc1/Rp16v8+Ou1x67Kq8jbYn9B70klN2hOyGT1y4TOTb5GI38waoxeOn+
IzDf3DFQG3id5Z+WknuO9wH0dKhPWWxzsvDywIqi6J6+oo1vp+6LAtzVOEtFQE7vO+mK715FafIhh
xJalx15kVlcNDSXC0S66GWYzJNrZ59PMGvFDuVWyI9pE5yRFXl+o7Yy9uBgVCUdPMLLYKQ4sZ0/
EJGqS+m/yJ8GnbSOgBwAA";
            rci.setCryptKeyPair(publicKey, secretKey);

            System.out.println("before " + new Date());
            System.out.println("ログイン認証中");
            rci.authUser("山田次郎", "pass1", "SHA-256", 0);
            System.out.println("after " + new Date());

            // シングルトン
            Packet single = rci.invoke("single", "example.test.SingletonSample", "getInstance");
            single = rci.exec(single, "getCount");
            int count = 0;
            count = single.toResult(count);
            System.out.println("example.test.SingletonSample.getCount():"+count);

            System.out.println("count を+10 に更新");
            count = count + 10;
            single = rci.exec(single, "setCount", count);
            single = rci.exec(single, "getCount");
            System.out.println("example.test.SingletonSample.getCount():"+single.getResult());

            // インスタンスの破棄
            single = rci.exec(single);

            // ソケット切断
            rci.disconnect();
        }
        catch (Exception e) {
            e.printStackTrace();
            System.exit(3);
        }
    }
}
```

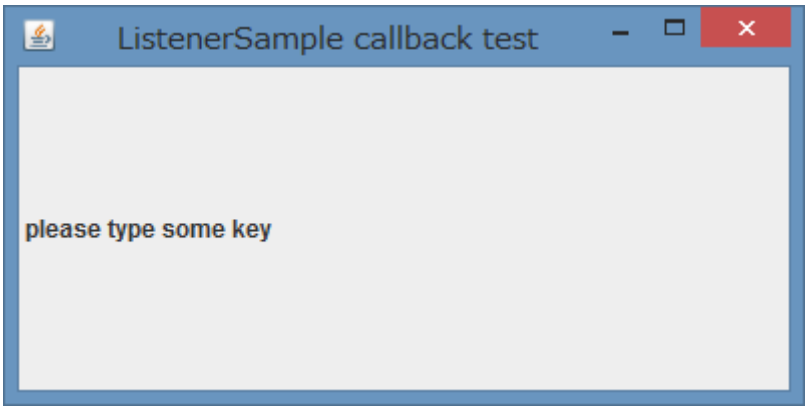
■サーバからの非同期イベント受信サンプル

javax.swing.JFrame をサーバで起動しキーイベントをクライアントで受け取ることができる。

サンプルプログラム一覧

No.	クラス	概要
1	example.test.Test2	RCI クライアントのメイン
2	example.test.ListenerSample	JFrame をサーバで起動する
3	example.test.ServerKeyEvent	サーバでイベント受信しクライアントへ送る
4	example.test.ClientKeyEvent	送られたイベントをクライアントで受け取る

RCI サーバを起動した状態で、example.test.Test2 を実行すると ListenerSample をクライアントから起動しサーバ側で以下のような JFrame が表示される。



＜RCI クライアントサンプルメインプログラム（example.test.Test2）＞

```
public static void main(String[] args) {
    String server = "localhost";
    int port = 5243;
    Rci rci = new Rci();
    try {
        rci.connect(server, port);
        rci.authUser("山田太郎", "pass", "SHA-256", 0);

        // サーバで JFrame を起動するクラスのインスタンスを生成する。
        Packet listenerSample = rci.invoke("new", "example.test.ListenerSample");

        // JFrame を表示するメソッドを実行する。
```

```
            listenerSample = rci.exec(listenerSample,   "showFrame");

            // クライアントとサーバでイベントを受け取るクラスを登録する。
            ClientKeyEvent cke = new ClientKeyEvent();
            ServerKeyEvent ske = new ServerKeyEvent(); // このクラス実体はクライアントにも必要
            Class<?>[] paramTypes0 = {java.awt.event.KeyListener.class};
            listenerSample =
                rci.registListener(listenerSample,   paramTypes0, cke, "addKeyListener", ske);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

＜JFrame をサーバで起動する（example.test.ListenerSample）＞

```
package example.test;

import java.awt.event.KeyListener;

import javax.swing.JFrame;
import javax.swing.JLabel;

public class ListenerSample {
    private JFrame frame;

    public void showFrame() {
        frame = new JFrame("ListenerSample callback test");
        frame.setBounds(200, 200, 200, 160);
        frame.setSize(400, 200);
        frame.add(new JLabel(" please type some key "));
        frame.setVisible(true);
    }

    public void addKeyListener(KeyListener l) {
        frame.addKeyListener(l);
    }
}
```

キータイプするとサーバで ServerKeyEvent が受け取ったイベントを以下のように標準出力に出力し、クライアントに転送する。

java.awt.event.KeyEvent[KEY_PRESSED,keyCode=65,keyText=A,keyChar='a',keyLocation=KEY_LOCATION_STANDARD,rawCode=65,primaryLevelUnicode=97,scancode=30,extendedKeyCode=0x41] on frame0

java.awt.event.KeyEvent[KEY_TYPED,keyCode=0,keyText=不明 keyCode: 0x0,keyChar='a',keyLocation=KEY_LOCATION_UNKNOWN,rawCode=0,primaryLevelUnicode=0,scancode=0,extendedKeyCode=0x0] on frame0

java.awt.event.KeyEvent[KEY_RELEASED,keyCode=65,keyText=A,keyChar='a',keyLocation=KEY_LOCATION_STANDARD,rawCode=65,primaryLevelUnicode=97,scancode=30,extendedKeyCode=0x41] on frame0

＜サーバでのイベント受信、転送（example.test. ServerKeyEvent）＞

```
package example.test;
import java.awt.event.KeyListener;
import com.graveng.rci.server.RciListenerInvoker;

public class ServerKeyEvent implements KeyListener {
    @Override
    public void keyTyped(java.awt.event.KeyEvent e) {
        System.out.println(e);
        RciListenerInvoker.gi().invokeClientListener(hashCode(), "keyTyped", e);
    }

    @Override
    public void keyPressed(java.awt.event.KeyEvent e) {
        System.out.println(e);
        RciListenerInvoker.gi().invokeClientListener(hashCode(), "keyPressed", e);
    }

    @Override
    public void keyReleased(java.awt.event.KeyEvent e) {
        System.out.println(e);
        RciListenerInvoker.gi().invokeClientListener(hashCode(), "keyReleased", e);
    }
}
```

クライアントでは ClientKeyEvent が起動され、受け取ったイベントを以下のように標準出力に出力している。(シリアライズして転送されているが、サーバでのものと同じではなく、情報がかなり欠落している。ローカルで実行する場合とは異なる。)

java.awt.event.KeyEvent[KEY_PRESSED,keyCode=65,keyText=A,keyChar='a',keyLocation=KEY_LOCATION_STANDARD,rawCode=0,primaryLevelUnicode=0,scancode=0,extendedKeyCode=0x0] on null

java.awt.event.KeyEvent[KEY_TYPED,keyCode=0,keyText=不明 keyCode: 0x0,keyChar='a',keyLocation=KEY_LOCATION_UNKNOWN,rawCode=0,primaryLevelUnicode=0,scancode=0,extendedKeyCode=0x0] on null

java.awt.event.KeyEvent[KEY_RELEASED,keyCode=65,keyText=A,keyChar='a',keyLocation=KEY_LOCATION_STANDARD,rawCode=0,primaryLevelUnicode=0,scancode=0,extendedKeyCode=0x0] on null

＜クライアントでのイベント受信（example.test. ClientKeyEvent）＞

```
package example.test;
import java.awt.event.KeyListener;

public class ClientKeyEvent implements KeyListener {
    @Override
    public void keyTyped(java.awt.event.KeyEvent e) {
        System.out.println(e);
    }

    @Override
    public void keyPressed(java.awt.event.KeyEvent e) {
        System.out.println(e);
    }

    @Override
    public void keyReleased(java.awt.event.KeyEvent e) {
        System.out.println(e);
    }
}
```

■サーバからの非同期イベント受信サンプル２（GUI 部品クラスの直接起動）

前項では JFrame の起動用のクラス example.test.ListenerSample を使って JFrame を生成していたが、クライアントから JFrame を遠隔生成して起動することもできる。

＜RCI クライアントサンプルメインプログラム（example.test.Test21）＞

```
public class Test21 {
    public static void main(String[] args) {
        String server = "localhost";
        int port = 5243;

        Rci rci = new Rci();
        try {
            rci.connect(server, port);
            rci.authUser("山田太郎", "pass", "SHA-256", 0);

            Packet jframe = rci.invoke("new", "javax.swing.JFrame", "GUI sample callback test");
            rci.exec(jframe, "setBounds", 200, 200, 200, 160);
            rci.exec(jframe, "setSize", 400, 200);
            rci.exec(jframe, "add", new JLabel(" please type some key "));
            rci.exec(jframe, "setVisible", true);

            ClientKeyEvent cke = new ClientKeyEvent();
            ServerKeyEvent ske = new ServerKeyEvent();
            rci.registListener(jframe,   cke, "addKeyListener", ske);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

■ログ出力方法（非 Android 環境でのログ）

デフォルトのログファイルは logfile.txt でログレベルは INFO になっている。
ログは以下の場所に出る。
“user.dir”/log/logfile.txt
※user.dir は System.getProperty(“user.dir”)で取得できるディレクトリ。通常カレントディレクトリになる。

(1) サーバでの設定方法
「ServerConfig.xml　// RCI サーバの設定ファイル例」を参照

(2) クライアントでの設定方法
以下の環境変数を設定すればログファイル名とログレベルを変更できる
com.graveng.rci.util.Logger.logFileName
com.graveng.rci.util.Logger.logLevel
logFileName はログファイル名を設定し、logLevel は VERBOSE, DEBUG, INFO, WARN, ERROR のいずれかを設定する。

■パスワードのハッシュ化方法

com.graveng.rci.crypt.HashCrypt の main に以下のパラメータを与えて実行すると文字列（パスワード）のハッシュ化ができる。

args[0]:ハッシュ化する文字列（１６進数形式も文字列としてハッシュ化する）
args[1]:ハッシュ化の繰り返し回数
args[2]:ハッシュアルゴリズム名（MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512 のいずれか、省略時は SHA-256）

パラメータが以下ならば、
pass1 1 SHA-256
以下のようにハッシュ化されたものが出力される
e6c3da5b206634d7f3f3586d747ffdb36b5c675757b380c6a5fe5c570c714349

■秘密鍵と公開鍵の生成方法

com.graveng.rci.crypt.RSAKeyPairCrypt の main をパラメータなしで実行すると秘密鍵と公開鍵を生成できる。

com.graveng.rci.client.Rci の setCryptKeyPair で、ここで生成した公開鍵と秘密鍵を設定できる。サーバのユーザ設定ファイルに公開鍵を設定しておけば、ログイン時にパスワードの検証のみではなく、公開鍵の一致判定も行うので、より強固なセキュリティになる。(setCryptKeyPair で公開鍵と秘密鍵を設定しなければその都度生成したものが利用される。)

---------- 以下実行結果例 ----------

【KeyPairの生成】

Wed Mar 28 14:50:54 GMT+09:00 2018

---生成中...---

Wed Mar 28 14:50:57 GMT+09:00 2018

公開鍵 ＞

H4sIAAAAAAAAAFvzloG1uIjBLr8oXS8pvzQvuTI5sbgkJ1UvKzkxKzlVr6AovywzJbVIL7G4Mjc3taQoM1mvqDhRz
8k5KNgxoDQpJzPZO7VSVSmLL/pXTgszA5MPA3tufkppTmlxCYOYT1ZiWaJ+bmJJhr5TZrpnXklqemqRtQ8DXw
FYp2tFQX5eal5JIUMdA2NFAdAdIiANeiANeggNPX/my6+0/i3LzMDmycCRlFniDHRoiScDJ5Dpk5qXXpLhySCclll
UXOKXn1eVWpTvVFmS6lea68nAk5NfnlpcEpxa4pQJ1MBWnJmeV5obzcCZm5iel1lSmpJawsAU7VRRxCAAtjgn
MS9dD6gzKbWobc1UWe4pD7qZGBgqCv4jwD8QZmBgYCwtAmld81n8BxtHyAOwMqAow/rN31dE5TR5tibq3F+v
NcNi3hmb+rqjr5/O+PzCSlm/2vSv/dcpZleuOXd+/yh98cD89xPZXkTWc79K2nlMr/W1VMKM4CrGdoHSm1NNpl2+
ue3MKfFF3y3nVJ4XrptyNrG1+n3s6a2u125ZSmXEL21O4Dhz9tTr7Defw75acvuUmLQecJ3wcndw+6GLXmWuVtr
1bCKzXaKNT54/1K720Np96+GYxkU33m1nerT6zZUPjho+ru9jDz1qVmLQmFMf4Oqx6vpO2a0PQo7sv3wnNq89/J
78rL4Zj3x+3pU7F31LtzfTK55n98IblsH8ZhIFt+dO9/X8dCO9+viVU2IPMxNM9JwFZWb9O/Noz5WbkpEVxaDYZM
YaYiAZdiCLmREY3RUA7I849XsCAAA=

秘密鍵 ＞

H4sIAAAAAAAAAKVUezRUeRy/MxPrFUkhj1QUmzUk0VAJDRkSBnmk5hrXuJiHO3cwk6gkKVYbYr02nOgxjG2j
h8qrPMojRaEU0kYz6zXbosQO226nPfuHc/Z3zu+c3+9zP5/z/Xw+v3Mu/zdAio0AtkyEhg9gchhULhVko2EQPoQKhlA
hPAthRsCBEIIH2Vw6HUIRmIpH2CDe1s6dbOOKwBEgCtkhqBPEDbuV0Iple+KxgLQzoEhFUDsmFBQEU2GIga
KAqnMIGAEa00E02NgWpjkyUIgGIVYSJguB6RAxisVkSIiu4UAMgPk36vYZlV5AXb+6uf2j4ASEwdS/JQtoFAJY/5
9gklT2Qtnqb9pstHAA1hn4hs4M5IRx2J9HKrH+on09kyWpc8V8WPx8WPyXsEkzP+lcsvqojQOkHQGZAFhSEIeBO
gKykqMzxKChwY6AShCMsFEXJoMHIUxbLgq5cOiOgEIYMxJio2QItYUlAmk2TGNw6H6ALB2kMWCUEwihAN
bPVhJ32cLgMJBBw0uUARBygp+uLZ/WdxoLAFGsuS9rdn4DAIDhIPNSvlhtSlrGo2+BJkEBwS+TF33DjjnGg9+9E
mzM3JrXvC025r7wTaZ4yFLX+NCWT9bv08w7ntolTo5rtlf/NPqD9JBPrPy7gJv1+HihBiWTzMMkLON0p5ude9Jd0
fxArWCSkMNtU4lJawHjD436PywnPu0haAQfvBBHkWlueSAMFYm93hPknVGz+GrimeFKckJtOymCaGkYK70ie

5ff5qa22oQN/VYO5XX7jxZ0jVzHDlwRdYzZGDgTR/1rB+LWAQY5sa7E3Zef3dQu7/O4V/XkhT8jYd9LnazvMwecp3
tXt/r1GCXDpIMKlfldBLKSuTrreW7GHseJLtqhho4Hqv0wxQxvp6yVNds8cLuje5VP1MIjL/nPxua/yMyXhGGlKI+r
3E5MERD0hxRPkup55LHUIz5XeBP7Bfd3yXqGvHtkTS4ua6l9WWSpvebVSG7mj4NNTyv0J0bvNHSvczN18WjOp
N3UiSbFSr32b1lZ39mKbXoazh+xyBASLjSQPcI+PD7eu01oIMUz1HDw8yCTjy07SenDNQoylqcfvs3c/sy4R1HVrb1
EU/eSFPdNVeLxXEHjp75S0U119zeT4VU/Gz8sUTfCb9gmr5uT/lwFpT6ITvDzSutz2Od1C8uwlMsoPXdqWma2sOt
GFEOnoX9EheJZZMLzaDfFxO5k+Km4mBS7p+S/TbZgOXnvc50Oh0jLeXWzR40V3up9uMY8FUEhRi2iJeDItz5P7
A9rnVXNjNtlProe8FEwN9GGWWoX/3ilNsQ7b0jqeYHQrs6IgwkhFrjWKWmsMunN1mp4xzVktdZv46GVxQMlAs
V7Kd/TUnFt4XWs+9fWHLI0p0Rn7ylrtzcOvVtkueF6d96YddzVj3IE73jZPLFwZ3Fp8cGgvIL7+j0nBI0yizLZ6Xdrvb
WouS1dd1/qD6+jd99Vv95p3O62bvustfJoEtEgZhlIr5OxcQFjNidGttL6HxtdTi8OpeyY0DJw59PKjvjfe15TsLEsZ4yu
uR1TeVnnQD7/d5zsixqpUbK49+HeEo1nKd5JInV8F6NsCJMd7nWxLc1UZ8xRbmlN9t7jFfblizIpV5vZtHlyyuFqxtz
wSjB0iqQnL96jtpLGsR2+YGT0XDRdut/Bu9qralUc+cSJKCrDrbFWfkR/Y+BMf+3usux6+7Xd3qnpZu/OeSpswd4xK
sg6WzLYV9cZq2l2+HGz792sHHtv1blsJ1Gl0Y3A+F77vFOd0+9P07PFpR0vd0bLNXWvWpTJyTuqZ5JxrlxFu8LI3s
NqoiWn2jetTtyZH/GrKRovPJIMnrnVhA4yV22nja7Z0aakt1JsZWgaXwdxcbMkpfwLqsnBJ6c0U7w/hPOXE47rDSi3
aDM3lQ57cKty11avaSa+zy2g+M4Etaf2jGtoEdSSC3PWVSiDZ+Ou3Mll1mApJtRFmSxPEQoK7deHsJzkfONmRL2
90Ut1L+kj5z6uz1d7t7TRacc10zlcsCbYAclzUFOL1gabg9Em1zEZSawmJQtRSGGkqCh0ortmNrCfhOt8FuAb13OS
93aw3qWlYu3rQevzVXNWfD3Mo9N89xjDIt75JN9P45FNB7ILejxrsoYd9qqqLMYkTvKDifoTaZSnb6AHAAA=

【生成したKeyPairの検証】

【平文】

＜RSAKeyPair暗号化の動作確認＞

【秘密鍵で暗号化】

H4sIAAAAAAAAAAEbAeT+rO0ABXVyAAJbQqzzF/gGCFTgAgAAeHAAAAEAOjNHJCmsFh1JoI1Sa6slVFuB/Fg
CKUTvTAwLCf3mOKC3HFR6R/IWvaKZ1Gpo9SgoujrygZYJXgk8FC3U1n8FL6+yDh0diR8sokcJd2DozIFtWYWg3
sF9BoEdWMy16Iawb54H/XT7cLsIriEtzYSSlQw6qfQ1DzIG4p8E8VVAThr+mBRreT8CGcC8ZaxXzh7kGcT9e2iKA
Id/H4mCB8PC5ztFjAImTh7AnlIl0W2RtjTHlCYww67oEgHbAVHEabgK3TvMHBgjcuspHfS1gGgGRkUsYVwCJqy
37aRPvSweq0S9dOqRcIwYwBz3RYRmixNJJr1caT4cdhcJhgWLgwq7QYkM/hwbAQAA

【公開鍵で復号化】

＜RSAKeyPair暗号化の動作確認＞

【公開鍵で暗号化】

H4sIAAAAAAAAAAEbAeT+rO0ABXVyAAJbQqzzF/gGCFTgAgAAeHAAAAEAhFXNx205Pfl2/PadLcoBTRX2iEPw
pyQxiTj+ZewkYg44Yb7TGf1WHPSCoVuSwiUMRGpaedVOvdfFyXlXFYART7U0RmprwIO69oR1XZ9wYZkMReC6
lMwCXP8MEu96dNXATmTszSsoQJ2SNiY/eNMuSqn3t8EzMmJCY56u549tqfVje8/gKPuTtnuGfeaRkCakIk45ppIj
w/r3O89ES2+2l/7ZCq4H15/6Co+3kuzmIY5OOkvVdII/f0QUje8gYAaDxRhJurXl+M9QSM4b91NRX82JKasZoGiwfq
F0tHWxgA/rmE0IVaToomOqcg6dPfdJ6MqlGVFLD2OWzSfYucodx62Ke6EbAQAA

【秘密鍵で復号化】

＜RSAKeyPair暗号化の動作確認＞

■鍵交換方式

RCI クライアントが com.graveng.rci.client.Rci の invokeExchangeKey()を実行するとクライアントで生成した公開鍵がサーバに送られ、サーバで生成した AesCBC128 の共通鍵と初期ベクターが公開鍵で暗号化され、Base64 にエンコードし、クライアントに送られる。クライアントでは受け取った共通鍵と初期ベクターを秘密鍵で復号し、以降、パケット全体を共通鍵で暗号化した通信が行われる。(共通鍵での暗号化が必要になるのは、RSA キーペアによる暗号化は処理性能的にコストが高い点と 117 バイトまでしか 1 度に暗号化できないからである。公開鍵が公開されてよいのは、公開鍵で暗号化したものは対になる秘密鍵で復号することはできても、公開鍵では復号できないからである。ssh や https のように暗号化アルゴリズムをクライアントとサーバでネゴシエーションしていないのは、クライアントとサーバの開発元が RCI では同じなので、その必要性がないからである。)

以下は電文例。

・クライアントからの要求電文例

<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_92" class="java.beans.XMLDecoder">
 <object class="com.graveng.rci.Packet">
  <void property="params">
   <array class="java.lang.Object" length="2">
    <void index="0">
     <string>ExchangeKey</string>
    </void>
    <void index="1">

公開鍵

<string>H4sIAAAAAAAAAFvzloG1uIjBLr8oXS8pvzQvuTI5sbgkJ1UvKzkxKzlVr6Aovy
wzJbVIL7G4Mjc3taQoM1mvqDhRz8k5KNgxoDQpJzPZO7VSVSmLL/pXTgszA5MPA3t
ufkppTmlxCYOYT1ZiWaJ+bmJJhr5TZrpnXklqemqRtQ8DXwFYp2tFQX5eal5JIUMdA
2NFAdAdIiANeiANeggNPX/my6+0/i3LzMDmycCRlFniDHRoiScDJ5Dpk5qXXpLhySCc
lllUXOKXn1eVWpTvVFmS6lea68nAk5NfnlpcEpxa4pQJ1MBWnJmeV5obzcCZm5iel1l
SmpJawsAU7VRRxCAAtjgnMS9dD6gzKbWobc1UWe4pD7qZGBgqCv4jwD8QZmBgYC
wtAmld81n8BxtHyAOwMqAow2STd8xOM7gOrw78qrJph8Ztk65tqyv+ac30q9YsEpd7fr
a2w92twHX6tZnVWc7dzybVRxx7+vvs1ekxrIWdSdEl/Nbzl3rzZz6vlzaqqWAOEu3cI1a5
7cxrH+UFnxYf9+hV/7F2yy+VryaP/vbLPDS13jV1W+mH3x3BkZXdUT3rgwI+5bvNDT6
1mXNhfqEJp1HHmbV/9+mdYNiZ+t7bm1OfrS/k0tXavRfjn55/V2Vdc68ldE2ZWT7HpKt
MW4742cywz9yoOGXn1H/7t99jLRTNnNgSlhywSdl1fqSPRlXNk2v3/J2L3Le3VbdcDGJ
fv7DOU0TW4Zdp0cIQbv16Ix5rE07lJ003+aNvnLpYUQyKTWasIQaSYQeymBmB0V0B

AF4Ug9V7AgAA</string>
      </void>
     </array>
    </void>
    <void property="serializedParams">
     <array class="java.lang.String" length="2"/>
    </void>
  </object>
</java>

・サーバからの応答電文例

<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0_92" class="java.beans.XMLDecoder">
 <object class="com.graveng.rci.Packet">
  <void property="params">
   <array class="java.lang.Object" length="2">
    <void index="0">
     <string>ExchangeKey</string>
    </void>
    <void index="1">
<string>H4sIAAAAAAAAAFvzloG1uIjBLr8oXS8pvzQvuTI5sbgkJ1UvKzkxKzlVr6Aovy
wzJbVIL7G4Mjc3taQoM1mvqDhRz8k5KNgxoDQpJzPZO7VSVSmLL/pXTgszA5MPA3t
ufkppTmlxCYOYT1ZiWaJ+bmJJhr5TZrpnXklqemqRtQ8DXwFYp2tFQX5eal5JIUMdA
2NFAdAdIiANeiANeggNPX/my6+0/i3LzMDmycCRlFniDHRoiScDJ5Dpk5qXXpLhySCc
lllUXOKXn1eVWpTvVFmS6lea68nAk5NfnlpcEpxa4pQJ1MBWnJmeV5obzcCZm5iel1l
SmpJawsAU7VRRxCAAtjgnMS9dD6gzKbWobc1UWe4pD7qZGBgqCv4jwD8QZmBgYC
wtAmld81n8BxtHyAOwMqAow2STd8xOM7gOrw78qrJph8Ztk65tqyv+ac30q9YsEpd7fr
a2w92twHX6tZnVWc7dzybVRxx7+vvs1ekxrIWdSdEl/Nbzl3rzZz6vlzaqqWAOEu3cI1a5
7cxrH+UFnxYf9+hV/7F2yy+VryaP/vbLPDS13jV1W+mH3x3BkZXdUT3rgwI+5bvNDT6
1mXNhfqEJp1HHmbV/9+mdYNiZ+t7bm1OfrS/k0tXavRfjn55/V2Vdc68ldE2ZWT7HpKt
MW4742cywz9yoOGXn1H/7t99jLRTNnNgSlhywSdl1fqSPRlXNk2v3/J2L3Le3VbdcDGJ
fv7DOU0TW4Zdp0cIQbv16Ix5rE07lJ003+aNvnLpYUQyKTWasIQaSYQeymBmB0V0B
AF4Ug9V7AgAA</string>
     </void>
    </array>
   </void>

```
   <void property="result">
    <array class="java.lang.String" length="2">
     <void index="0">
```

公開鍵で暗号化された共通鍵を Base64 でエンコードしたもの

```
<string>A+gNG8t0aQSLWobYPVFE68XUGUES0835cgI3fbbB7eyUmcin10aCcZyMYU
FV+JPcqNTXC2339MLSkngWN7zDkzWlBlt0y4rTWP8Yr0azmgNumAh2z9QrITMCyF
iNmWUq6IF6YvCwaphtbmN+xtbePvwYZYjkXDv8vq3z/ZaxQEw9406ejhpyiLQPeIfWc
Gou62+n9sH4WR6Ss3JhuodQaHTKTMvnv7dAAuJq86cDctcd6BpBWCcDrdx72BhWPx
v/yuTGYiRrBAMDA9gwn0/S1sL1WCJy9QuydrpD0accdF0NLax7Ogk+6FIR7bs6BEGgf
OdbmPnh3sgphz1DB8uLHQ==</string>
```

```
     </void>
     <void index="1">
```

公開鍵で暗号化された初期ベクターを Base64 でエンコードしたもの

```
<string>aG5Srjsa9QYzEMvATV1t2y1QcJw9vLSdGDhG1C7p6ns8JcBFDHttfPkvb9Vg
hBHXy6p5SpQlzlFFbSg/LKSL6f1wlWtxfgeyk9VKhdZZF7E+EFAklMhlY752ynTiEgIL
U2kjvciPcn/udhSHPokbuJ8mbfxN38ZdOcJEJFhbdJfX1Jsn/+Zght7nKXev4hyCxxIPbD/
7T1LrPLfHdfnqjlex2EsY62eRZjXpmwpw7ZPlxGJ4M9je8IPCN5VhkZr5pj6E+kI8aIqx4
QWB5OkKdbEs+nCLpAwmgO/palGLlysL7+hZPwT0xpp1Nyih2L+EAGPmtM0N4VVM
/0odVQM2Ow==</string>
```

```
     </void>
    </array>
   </void>
   <void property="serializedParams">
    <array class="java.lang.String" length="2"/>
   </void>
  </object>
</java>
```

■ユーザ自身によるパスワードの変更

com.graveng.rci.client.Rci の invokeUpdatePassword で RCI クライアントユーザ自身によるパスワード変更や公開鍵の登録ができる。元の設定ファイルは”ユーザ名.xml.bak”にコピーされる。

■ログイン試行対策

総当たり攻撃のようなアタックをされた場合の対策として、com.graveng.rci.server の LoginFailureUser により、A 時間内に B 回数ログインに失敗した場合は C 時間ログインできないようにする機能がある。

デフォルト値は、

A: 1 時間

B: 10 回

C: ３０分

としている。

変更するには、ServerConfig.xml に以下を追加する。

```
<void property="loginWatchingTimeMillis">
 <long>36000000</long>
</void>
<void property="loginLimitFailCount">
 <int>10</int>
</void>
<void property="loginRejectTimeMillis">
 <long>18000000</long>
</void>
```

第三者の嫌がらせによっても正規ユーザがログインできなくなるので、このような機能を無効にしたければ、loginRejectTimeMillis を 0 にする。(公開鍵を登録する十分なセキュリティが確保されたログイン設定ならば、攻撃者は対応する秘密鍵を持っていなければならず、不正に侵入されることはほとんどありえない。[どんな対策をしても少なくとも正規ユーザはログイン可能なのだから、確率的にはゼロではない。])

■クライアントのタイムアウトの設定

メソッドコールのみでサーバからのイベント非同期通知機能を使わない場合は、Rci の isUseEventListener=false [rci.setUseEventListener(false)]とすれば、要求に対する応答は無限待ちになる。無駄に CPU を使わないので効率は良い。（コネクションは切れなくても何等かの不具合でサーバが応答を返さないことはあり得るので、この場合は無限に待ってしまうことになる。）

isUseEventListener=true（デフォルト）とし、イベント非同期通知機能を使う場合は、Rci.java ファイル内の Invoker クラスの以下のデフォルト設定で小刻みに応答待ちループをしている。（同じソケットで要求に対する応答と非同期通知を同時に受けるようにするため。遠隔メソッドコールの際に受信を待たずに応答を受けたらイベントリスナーをコールする非同期方式にする方法もあるが、普通のプログラムの作りでは、メソッドコールはメソッド内で応答を待つので、イベントリスナー方式は不自然なため採用していない。）

```
long recvSleepMs = 10; //  ミリ秒の応答待ち時間
int recvSleepRetryCount = 10000; //  応答待ち繰り返し最大数
boolean isIncrementalSleep = true; //  待ち回数が増えるごとにスリープ時間を増やす場合は true
```

これらの値を変更するには、connect 後に以下のようにする。

```
String server = "localhost";
int port = 5243;
Rci rci = new Rci();
rci.connect(server, port);

rci.getInvoker().setRecvSleepMs(100);
```

遠隔メソッドコール時にタイムアウトすると TimeoutException が送出される。

■ライセンスについて

ライセンスは Apache License, Version 2.0 としている。

いわゆるオープンソースのフリーソフトウェアで原則的に使用や配布は自由にできるが無保証である。

```
/*
 * Copyright (C) 2018 Gravity Engineering Corporation
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *         http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

■今後の展望

Spring フレームワークというものがあるが、一言で言えば、xml 形式によるダイナミックに変更できないクラスの検索条件がお粗末なリードオンリーのオブジェクトデータベースである。プログラムを大量の xml のテキストファイルに置き換えて本当に開発効率が上がったのだろうか。

オブジェクトデータベースはアプリケーションサーバではなく DB サーバにあることが望ましいが、既存のオブジェクトデータベースはリモートへの設置が困難である。オブジェクトがリモートにあるとオブジェクトをネットワーク越しに転送しなければならない。

RCI はオブジェクトをネットワーク越しに転送しているのであり、本格的なオブジェクトデータベースの基盤にもなり得る。オブジェクトは xml のテキストファイルにするのではなく、オブジェクトデータベースに格納しなければ、大昔のテキストファイルのプログラムを grep で調べてデバッグするようなことになる。データベースは検索／更新できるから、大量のデータを扱える。もし、データベースのデータがテキストファイルになっていれば、大量のデータを扱うのは難しい。

自動運転のような人工知能は認知、判断、行動を繰り返している。曖昧さがあるパターン認識が判断である。この判断はパターンクラスオブジェクトの検索のことであるから、オブジェクトデータベースの検索になる。そのオブジェクトデータベースがクラウド上にあれば、多くのパターンから検索が可能になり、より精度が高い判断が可能になる。

かなり以前から望まれているが、未だに現実的なものになっていないオブジェクトデータベースの開発は人工知能の発展に不可欠だろう。昔はプログラムとデータは分かれていた。プログラミング言語がオブジェクト指向になりプログラムとデータは一体化した。その次に訪れる進化はプログラム自体のデータ化で、これによりロジックはダイナミックに組み替えられるようになる。プログラムは小さな単位でデータベースに格納され、それぞれは単純になり、巨大になりがちな既存のシステムとは異なり、データベースの値を変更するようにメンテナンスは容易になる。巨大で複雑なシステムは簡単なものになる。

小さなプログラム（ジョブ）の連動的な動きを規定するジョブネットというものがあるが、連動的な動きを規定するのはハードコーディングしているのと同じであり、ロジックをダイナミックに組み替えられない。A ジョブが終了した状態が B1、B2 状態になるなら、B1 状態であることを検索して動く C ジョブのような方法でジョブはフラットに並べイベントドリブンのようにしなければならない。それで認知、判断、行動を繰り返すことになり、どんなロジックも表現できる。類似判定の方法論は枝葉のことであり、人工知能も普通のシステムもその構造のゴールは基本的に同じである。

それを実現するものがオブジェクトデータベースであり、データベースも進化が必要である。情報処理はデータ管理そのものであるから、古い設計思想のデータベースでは良いシステムは作れない。人工知能以前にまだやり残していることは多い。