Possíveis Melhorias para o projeto

Geral

- ~~Adicionar Gradle;~~

Possíveis melhorias para o App

- ~~Migrar de **actionbarsherlock** para **AppCompat** para resolver problema de compatibilidade com Android 7.0;~~

Possíveis melhorias para o Servidor

- Mudar banco de dados de MySQL para Cassandra;
    - ~~Ou atualizar MySQL de v5.2 para v5.6;~~
- ~~Atualizar Tomcat de versão 6 para versão 9;~~
- ~~Adicionar sistema de autenticação e permissão por usuário;~~
    - Criar interface para gerenciar isso;
- Tornar comunicação entre app e servidor segura;
- ~~Adicionar criptografia de senhas no banco de dados;~~

Possíveis melhorias para o Objeto Inteligente

- Realizar cadastramento automático;
- Realizar monitoramento do objeto inteligente (push ou pull?);
- ~~Atualizar RXTXComm;~~

O projeto atualmente faz a comunicação com o objeto inteligente a partir de um protocolo chamado modbus. È o que vamos usar, também, para realizar cadastramento automático do objeto e também o seu devido monitoramento.

Atualmente, um objeto inteligente é representado na base de dados pelas seguintes informações:

- **Nome** - que na verdade vira nome de uma **categoria**, por *wtf* razões;
- **ID** - pode ser automático, eu acho. Mas para isso ocorrer acredito que seja necessário repassar o ID gerado para o objeto inteligente novamente, de forma que ele possa lidar com as manipulações de maneira esperta;
- **URL** - **acho** que pode ser detectado automaticamente;
- **ID Modbus** - È a porta com o qual o dispositivo vai se comunicar. Pessoalmente, penso que um servidor pode atender vários objetos inteligentes, logo este parametro permite uma especie de

multiplexação a nivel de servidor. **Talvez** possa ser automático, sob as mesmas condições básicas do ID. Entretanto, pode dar treta na comunicação a partir do modbus (oras bolas, se o modbus usa esse campo, como caralhas ele vai ser automático? Talvez tenhamos que especificar na mão, mesmo);

Ou seja, tudo o que precisa ser especificado para cadastrar um **objeto** é nome e ID Modbus (que invariavelmente é passado durante a comunicação do protocolo). Mas não é só isso..

Uma vez que nada é de graça, não basta a gente inserir objetos. A gente tem que inserir também o que no sistema está classificado como **serviços**, que são **agrupamentos de parametros.** Para cadastrá-los, temos que ter em mãos as seguites informações:

- **ID do serviço** - é coletado automaticamente no cadastramento do serviço;
- **Nome** - precisa ser informado pelo usuário;
- **idservicemodbus** - É o function code do Modbus, usado pelo protocolo para indicar qual operação a requisição quer fazer. Teoricamente, pode ter valor 01, 02, 03, 04, 05, 06, 15 e 16, mas na base temos uns valores bem bizarrinhos como 00, por exemplo.
  Em tempo: Esse campo não me parece fazer muito sentido. A function code serve para descrever o que caralhas a requisição quer fazer, como se quer fazer apenas leitura ou escrita de dados. Num sistema com monitoramento isso não me faz o menor sentido, pois precisamos tanto **escrever** quanto **ler** dados, certo?
- **idcategory** - podemos pegar isso durante o cadastramento do objeto;
- **idregistermodbus** - ID do registrador que será comunicado pelo modbus. Note que isso é realmente só um identificador, e tenho 90% de certeza que isso é enviado junto no protocolo;

Ok, cadastrado objeto e serviços, precisamos cadastrar parametros. Todos os parametro tem as seguintes informações básicas:

- **Nome** - precisa ser informado manualmente;

- **Tipo** - precisa ser informado manualmente. È possívelmente um ENUM ou algo assim;

Todas as demais opções dependem do tipo do parametro, como valor máximo e minimo e opções disponiveis para o usuário escolher. Logo não a trataremos aqui.
Um detalhe é que, logo após, temos que fazer a associação entre o parametro e o serviço. Talvez na API vale deixar fazer tudo numa única sacada..
A principio. É isso. O parametro é a menor unidade do sistema e provavelmente o monitoramento vai funcionar ao configurar um valor para ele. Quanto as tarefas, falta:

- Entender modbus e dominar 100%;
- Implementar a porra do modbus certinho no java e no C++:
- Criar interfaces para a API;

Problemas atuais:
Investigando aqui o código e investigando o protocolo, notei algumas coisas:

- Modbus foi concebido para ser um protocolo **master-slave**, ou **cliente-servidor**, como preferir. Nos mesmos moldes do HTTP;
    - Isso deixa a seguinte pergunta no ar: Quem é, no nosso sistema, cliente, e quem é servidor?
    - Considerando o jeito como o sistema foi originalmente implementado, dá para considerar que o SOServer é o cliente e o EPOSMote é o servidor..
    - Mas, temos um problema: Temos que implementar tanto a adição **automática** de objetos quanto o **monitoramento** - que pode ser push ou pode ser pull.
    - São duas coisas conflitantes, que funcionariam provavelmente melhor se a comunicação do Modbus fosse master-master, pois assim seria perfeitamente viável que tanto o EPOSMote se comunicasse conforme sua própria vontade (e.g para se registrar, por exemplo) quanto o computador pudesse se comunicar com o EPOSMote sob sua própria vontade também (e.g para obter status, mandar comandos, etc);

    - Podemos manter a arquitetura atual de master-slave. Mas, com ela, precisamos fazer polling em um dos lados da arquitetura:
        - Do jeito que estamos hoje, precisamos fazer polling para descobrir **objetos inteligentes novos na rede**. Mas...como fazer polling para descobrir objetos inteligentes na rede senão temos sequer seu *address* ou informações que o modbus pede para localização?
        - Ou, se escolhermos o SOServer como servidor, precisamos que o EPOSMote faça polling para descobrir **quais comandos** executar;
    - Particularmente, penso numa abordagem mista, meio que *master-to-master* pelo menos na primeira parte do processo. Isso porque, como já deixado claro acima, temos o problema de descobrir objetos inteligentes novos na rede sem sequer saber seu *address*, mas particularmente isso é meio complicado..A menos que nós acabemos por definir um **endereço fixo** para o **dispositivo correspondente** ao Master (o que é até viável, se for parar para pensar..);

Protocolo

A gente tem que criar um protocolo para a comunicação do dispositivo do EPOS com o Java e vice-versa.
Este protocolo deverá usar os registros que o modbus fornece corretamente. Por exemplo:

- Holding Register para enviar/ler configuração;
- Input registers para armazenar status e medições - não sei se será usado;
- Coils para enviar/ler bits on/off;
- Discrete inputs para ler bits on/off - acho que não será usado;

Pessoalmente, acho que:

- O programa **deve** tanto ler quanto escrever configuração como comandos, logo, holding registers vão ser a entidade mais usada;
- Coils talvez consigam representar alguma coisa, no mesmo sentido que holding registers, mas para sequencias de bits.
- Outra opção seria usar holding registers para armazenar dados de status (ex.: que tabela/coluna está sendo processada), mas envolveria o uso de templaes logo é quase a mesma coisa, eu acho. A treta aqui é que temos campos bem definidos. Mas talvez vale checar.

No geral, temos o seguinte:

- Temos que ter uma forma, no protocolo, de passar strings ao SOServer. Tratando casos como tabela e campo a ser inserido, tamanho dos dados a serem processados.
- Temos problemas, com a questao da ordem das mensagens.
- Como o envio sera organizado? Quais registradores receberao as informaçoes e como essas informaçoes serao organizadas dentro dos registradores?
- Podemos ter

Apresentação

Precisamos explicar os componentes do trabalho e como eles estão conectados entre si, também.
O que a gente fez no trabalho:

- Criamos uma máquina virtual para desenvolvimento usando Vagrant - incluindo instalação de Android SDK para compilação de apps Android; **(E)**
- Criamos uma máquina virtual para uso em produção usando Packer - que permite criar máquinas para diferentes provedores facilmente; **(E)**
- Implementamos o Gradle em todos os módulos Java do projeto - que permite construir projeto sem o uso de uma IDE e sem configurações complicadas; **(E)**
- Atualizamos todas as dependências - como Tomcat, MySQL, MySQL client e Apache Commons - e listamos as bibliotecas Java como dependências do Gradle, de forma que não é necessário mais se preocupar com download de *jar* e afins;**(E)**
- Substituímos a biblioteca RXTXSerial - que fazia a comunicação do **SOServer** com o dispositivo - por um fork, chamado nrjavaserial, que é muito mais atualizado e permite o uso do projeto **sem a instalação de bibliotecas nativas** (no RXTXSerial era necessário instalar bibliotecas nativas manualmente..); **(F)**
- Fizemos uma implementação do **Modbus** tanto para Java quanto para C++, permitindo que ocorra interoperabilidade entre os sistemas, visto que um consegue se comunicar com o outro e validar a mensagem normalmente; **(F)**
- Fizemos uma API de alto nível para a comunicação do dispositivo inteligente com o **SOServer,** permitindo que o dispositivo altere status sob requisição do app Android e também informe status para fins de monitoramento**; (E)**
- Removemos o **actionbarsherlock** e o **slidingmenu** do app Android e substituímos pelo **Android Support** - que é uma biblioteca suportada pelo **Google**, que implementa **Material Design** e que suporta desde Android 2.3 até Android 7.0; **(F)**
- Modificamos a estrutura do banco de dados de forma que menos tabelas são consultadas, tornando o banco de dados mais simples como um todo e tornando o modelo de dados mais próximo do que é encontrado no aplicativo; **(F)**
- Montamos abstrações capazes de fazer nosso código em C++ rodar tanto em um computador como no EPOS II;

- Codificamos todas as senhas presentes no banco de dados com **bcrypt**, de forma que mesmo se o banco de dados for acessado não é possível saber qual a senha do usuário do sistema;**(E)**
- Implementado solicitação de dados do app do Android para o dispositivo inteligente, de forma que o app consegue obter o status do dispositivo no momento que lhe for preferido - deixar explícito a questão de pull;
- Iniciada implementação rústica do cadastro automático de objetos inteligentes, com a criação de classes responsáveis por fazer o cadastro desses objetos e afins (ainda não integrada à API do objeto inteligente, entretanto);