Universidade Federal do Amazonas
Manaus, 12 de Julho de 2017
Disciplina: Redes de Computadores
Profº.: César Melo
Equipe: ~~Matheus dos Santos Menezes~~
~~Marcelo Augusto de Lima Brasil~~
~~Lucas Dimitri Correia~~
Lucas Gonçalves Silva *
Willians Cassiano de Freitas Abreu

CheckPoint 3 (CP1) : Relatório da Aplicação

Aplicação: Um protocolo de aplicação para transporte de dados utilizando VANTs.

Contexto

Métodos de sensoriamento autônomo de áreas remotas, críticas ou perigosas tem sido utilizados em larga escala em projetos de pesquisa e comerciais para aumentar a eficiência no uso dos recursos humanos envolvidos. Dispositivos computacionais equipados com diversos sensores produzem dados importantes para o monitoramento dessas áreas, em um volume considerável. Métodos usuais de transporte de dados entre os sensores e entre os sensores e os interessados em geral são dificultados pelas condições ambientais envolvidas. Links de satélite ou transmissão direta dos dados podem ser inviáveis técnica e economicamente, além de aumentar o consumo de energia, um fator crítico em tais sistemas.

Solução

Esta aplicação busca possibilitar a transmissão de dados genéricos entre equipamentos sensoriais instalados em áreas remotas (i.e Raspberry Pi 3 com câmeras e microfones instalados na floresta), com Veículos Aéreos Não-Tripulados autônomos (i.e Quadrotor Asctec Pelican).

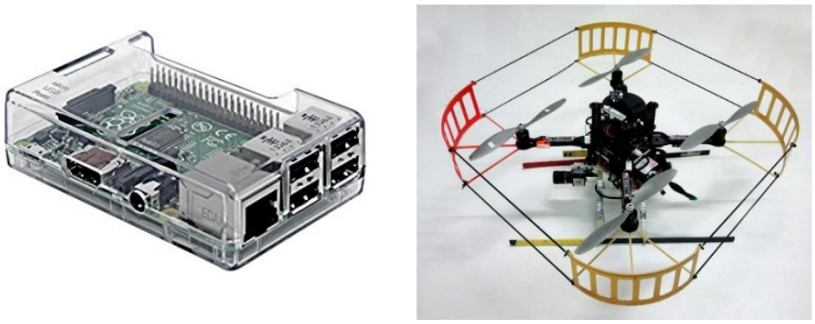

Figura 1: Raspberry Pi, à esquerda. Asctec Pelican, à direita.

Essa comunicação se dá através de uma rede adhoc entre os dispositivos. O Raspberry que possui os dados a serem transmitidos detecta a presença do VANT próximo, e inicia a transmissão de seus dados.

O Raspberry inicia a comunicação, e após o sucesso na conexão ao drone, compacta os arquivos na pasta especificada em um único e inicia a transmissão deste através da rede utilizando TCP. O drone recebe a stream do arquivo, e o armazena no disco local.

Finalizado o recebimento dos arquivos, o VANT retorna à base com os dados para os interessados, e ambos os equipamentos terminam a transação em um estado de "pronto" para a transação seguinte.

Requisitos Básicos

A aplicação deve garantir:

1. **Transferência de arquivos arbitrários:**
    - Transferência arquivos arbitrários entre os dispositivos, sem restrição de tamanho ou conteúdo.
2. **Integridade:**
    - Integridade dos dados recebidos pelo receiver.
3. **Segurança:**
    - Que os dispositivos finalizem todas as transações em um estado seguro, sendo bem-sucedidas ou não.
4. **Disponibilidade:**
    - Que a aplicação esteja pronta para uso assim que os dispositivos computacionais sejam inicializados, e ao final de cada transação, sendo elas bem-sucedidas ou não.

Requisitos Opcionais

Requisitos implementados opcionalmente:

1. **Aplicação como um serviço**
    - Implementar a aplicação como um serviço UNIX

Implementação

A interconexão entre os dispositivos é realizada alterando-se a configuração da interface de rede contida no arquivo `/etc/network/interfaces` presente em sistemas unix. A interface de rede sem fio dos dispositivos é inicializada em modo inet, estático e adhoc, e a cada dispositivo é atribuído um endereço IP estático. Desta forma, a detecção do ingresso de um dispositivo na rede é gerenciado pelo sistema operacional, e podemos transferir dados utilizando a API TCP.

Exemplo:

```
auto wlan0
iface wlan0 inet static
      address 192.168.1.1
      netmask 255.255.255.0
      wireless-channel 1
```

```
wireless-essid MYNETWORK
wireless-mode ad-hoc
```

Além disso, tornamos a configuração agnóstica a hardware. Os testes foram realizados utilizando um RaspberryPi 3 B e seu adaptador WiFi embarcado, comunicando-se com um computador de bordo AtomBoard no quadrotor Pelican, por meio de um *dongle* WiFi USB Asus N53.

Protocolo

1. Ao início de uma transação, o Raspberry Pi se encontra em modo de espera com o programa sender em execução, aguardando uma tentativa de conexão da parte do VANT. Ele detectará a conexão do VANT por meio da chamada `connect()` da API TCP, que retorna -1 enquanto a conexão não for bem sucedida, tentando novamente a cada 5 segundos.
2. O Drone por sua vez, está com o programa receiver em execução, travado na chamada ao procedimento `accept()` da API TCP , que bloqueia a execução do programa até que uma conexão por parte de um cliente à porta definida na constante `PORT` do arquivo receiver.c seja recebida.
3. Quando o drone está suficientemente perto do Raspberry Pi (A depender das limitações dos adaptadores WiFi utilizados) o *handshake* TCP é realizado, uma conexão é estabelecida, e os programas em ambos os sistemas finais são desbloqueados para continuar sua execução.
4. Com isso, o Raspberry inicia a compressão do conteúdo presente na pasta indicada pelo usuário na constante `BACKUP_FOLDER` definida no arquivo `sender.c`. O arquivo é comprimido com a aplicação unix `tar` e gera um arquivo com extensão `.tgz`. Este arquivo é nomeado usando a seguinte convenção: <BACKUP_FOLDER>_<unix_timestamp>.tar.gz e é armazenado na pasta acima de BACKUP_FOLDER. Unix_timestamp é a quantidade de segundos passados desde 1 de janeiro de 1970, de acordo com o horário do RaspberryPi. Este horário não é muito confiável, por se tratar de um sistema embarcado sem meios de sincronização em caso de uma reinicialização. O utilizamos para realizar distinção entre os documentos, caso sejam colhidos dados de uma mesma estação mais de uma vez e para manter a ordem cronológica entre estes.
5. Após isso é a realizada a leitura do arquivo resultante para a memória do dispositivo, fator pelo qual nossa aplicação está limitada pela quantidade total de memória principal disponível no dispositivo ou área de troca (swap), quando em utilização. O uso de área de troca não é usual em sistemas embarcados.

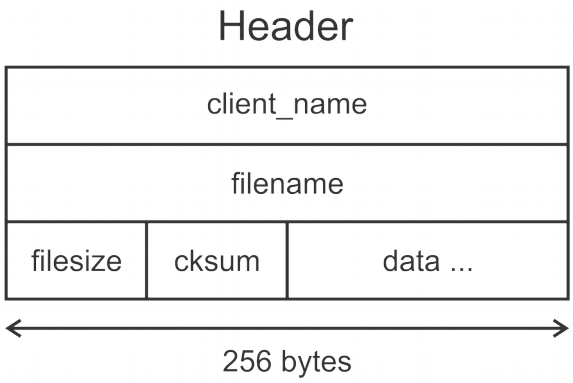

Figura 2: Modelo de cabeçalho

6. Com o arquivo em memória calculamos a soma de verificação que será inserida posteriormente no cabeçalho. A soma possui 32 bits e é calculada utilizando a hash não-criptográfica crc32b. Esta função foi escolhida pela sua facilidade de implementação e cálculo veloz.

7. O Raspberry então efetua a transmissão de um cabeçalho, descrito na Figura 1. Este cabeçalho possui um máximo de 640 bytes e inclui: O nome de host cliente (exemplo: `rasp_floresta_1`), o nome do arquivo que esta sendo transmitido (como descrito em 4), o tamanho do arquivo e a soma de verificação. O cabeçalho é codificado em um vetor de chars e transmitido pela rede.Quando estes dados chegam ao Pelican, este realiza o *parsing* das informações necessárias e as armazena.

8. Após isso, o Pelican cria uma pasta com o nome do host do Raspberry no diretório especificado pelo usuário na constante `ROOT_DIR` definida no arquivo `receiver.c`.

9. Com estes preparativos, a tranmissão dos dados pode ser iniciada, e a stream de bytes é enviada a partir do buffer que estava na memória a partir do passo 5. O envio é realizado utilizando o protocolo TCP, através das chamadas sendfile() e recv(), no RaspberryPi e no Pelican, respectivamente. O Drone salva os arquivos em um buffer na memória principal, estando limitado da mesma maneira que o Raspberry Pi pelas restrições descritas no passo 5.

10. Ao fim da transmissão a conexão é fechada em ambas as partes, e o arquivo é escrito da memória para o sistema de arquivos local no Pelican.

11. O passo 5 e 6 se repetem do lado do drone, ao final dos quais a soma de verificação resultante é comparada com a obtida no header do passo 7.

12. Arquivos temporários são apagados, memória liberada e os dispositivos retornam ao passo 1 e 2.

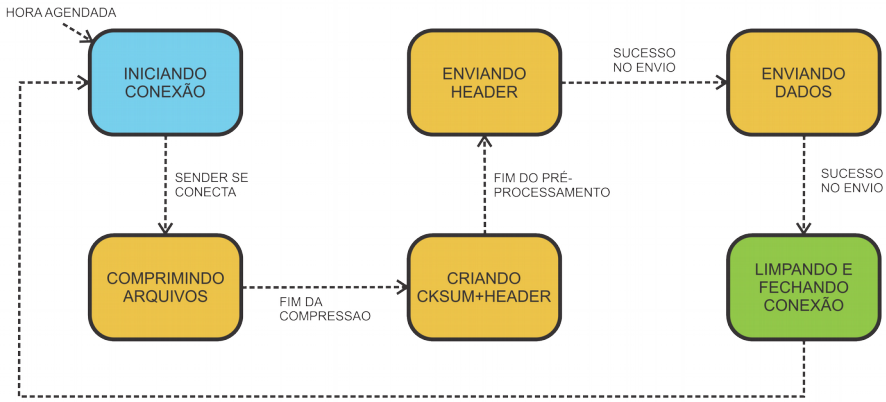

Figura 3: Máquina de estados, lado sender (RaspberryPi)

OBS: Em caso de falhas em qualquer um dos passos, quando indicado pelo retorno das funções de acesso ao sistema operacional, os arquivos temporários gerados são apagados, os procedimentos são reiniciados ao passo 1 e 2, em cada dispositivo.

O conjunto geral de ações é descrito de maneira compacta nas Figuras 2 e 3, e assumem-se transições implícitas de todos os estados para os estados em verde, em caso de excessões na execução.

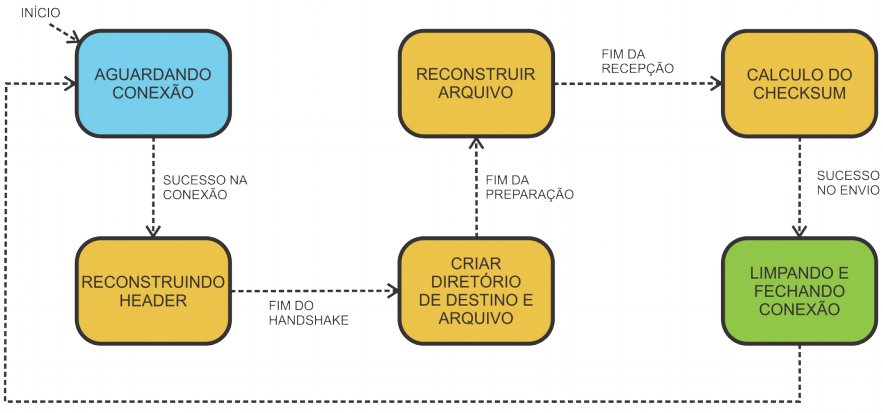

Figura 4: Máquina de estados, lado receiver (Pelican)

Arquitetura

Cliente Servidor:

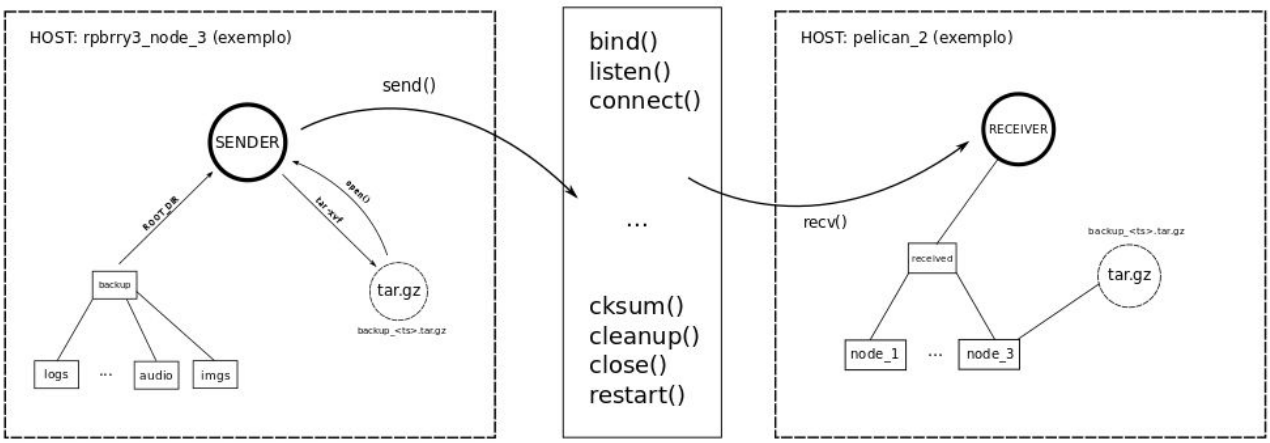

Figura 5: Descrição arquitetural do fluxo de dados na aplicação, e das interfaces utilizadas

A arquitetura utilizada foi a cliente-servidor, utilizamos a nomenclatura sender-receiver, pelo fato de que a conexão é iniciada pelo sender, o que o tornaria um cliente. Entretanto, a entidade sender é também a que possui os arquivos, o que a caracterizaria como um servidor.

Projeto

O diagrama UML incluso na figura 5 apresenta o padrão de projeto utilizado. Os arquivos são representados como classes e as dependências de `#include` são representadas como setas tracejadas, do dependente para a dependência.

As utilidades de arquivo como: compressão; criação de diretórios; leitura para a memória principal e outras estão separadas na classe FileUtils. As utilidades de protocolo, como: soma de verificação; debug; decodificação do Header e outras, estão contidos na classe ProtocolUtils.

A classe Header compreende o tipo de dado utilizado, compartilhado entre os programas. Enquanto as classes SenderSockets e ReceiverSockets encapsulam as chamadas ao sistema operacional, de maneira que os drivers Sender e Receiver sejam focados na lógica de aplicação.

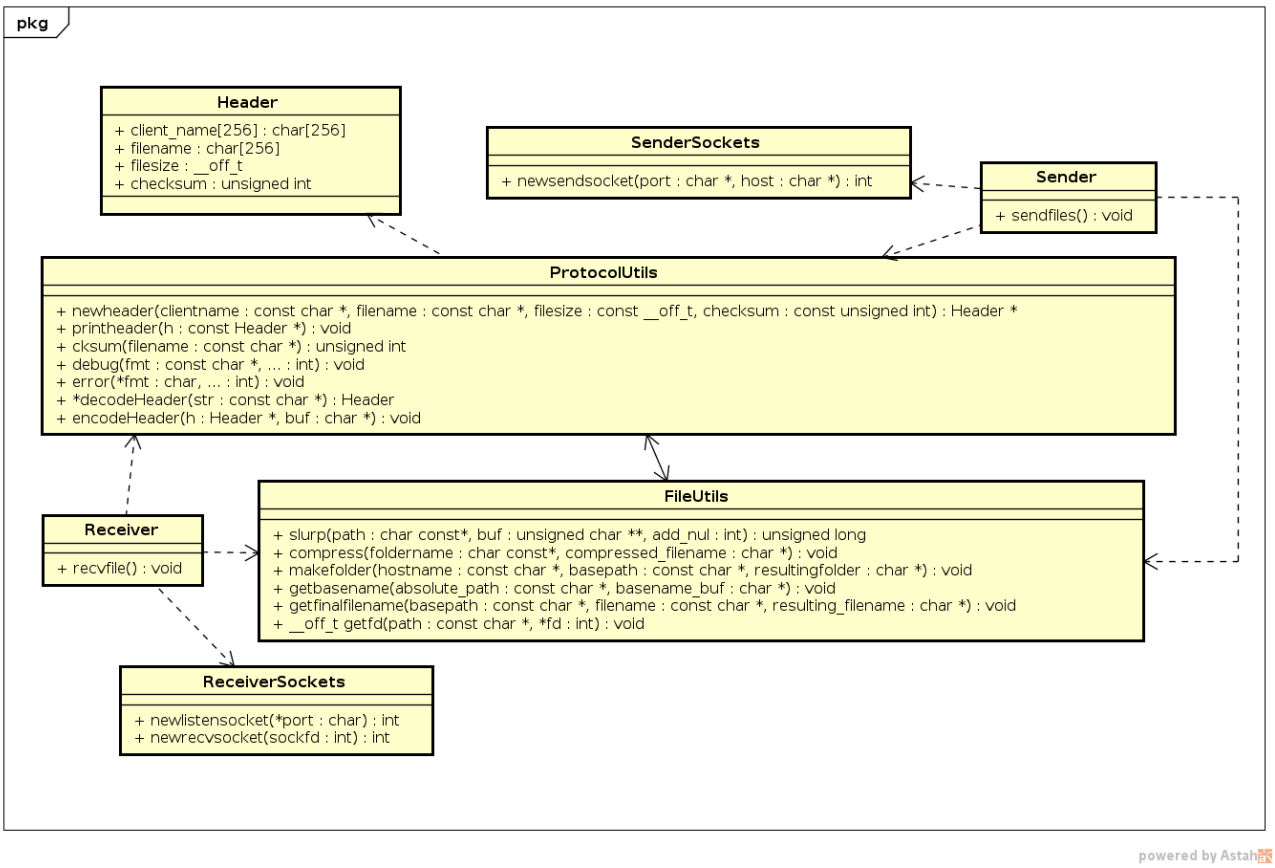

Figura 6: Diagrama de Classes representando o padrão de projeto

Sistema de Build e Estrutura do projeto

A implementação do projeto foi realizada em C, fato pelo qual o diagrama UML acima não captura totalmente a estrutura dos arquivos no disco, apenas suas depedências lógicas e funcionais. Separamos os arquivos de cabeçalhos (.h) dos arquivos de código (.c), que estão nas pastas include e src, respectivamente. Essa separação facilita o processo de manutenção e compilação, mantendo uma clara distinção entre interfaces e implementação.

Os arquivos de códigos estão separados por diretórios, sendo estes:

- **exec**: Contém os drivers para os binários sender, receiver e tester
- **receiver** e **sender**: Contêm a implementação da lógica da aplicação, assim como as chamadas à API TCP pelo sistema operacional dos programas sender e receiver.
- **utils**: Isola as funções compartilhadas entre as aplicações, como manipulação de arquivos, checksums e as estruturas utilizadas na comunicação.

Para a construção dos binários de execução e testes, utilizamos a aplicação UNIX make. Separamos os drivers das utilidades, de maneira que os códigos-objeto compilados das utilidades pudessem ser compartilhados durante o build entre os 3 binários.

No diretório src/exec, há os arquivos drivers, que possuem os main()’s para cada aplicação, estes drivers são “colados” em cima dos arquivos objetos compartilhados e linkados uns contra os outros, de maneira que a partir dos mesmos arquivos objetos, podemos gerar os 3 mains diferentes.

Após o final da compilação, são criados os arquivos de logs, que podem então ser escritos utilizando append pelas aplicações que deles se utilizam.

Adicionais

Decidimos implementar a aplicação como um serviço UNIX, e o caminho para isso foi a criação de 2 scripts em BASH para gerenciar a criação e o encerramento de instâncias dos programas no SENDER e no RECEIVER. Exemplo (run_forever_sender.bash):

```
while  [ true ]; do
  ps cax | grep sender > /dev/null
  if [ $? -ne 0 ]; then
    echo "Process is not running." | tee -a logs/sender.log
    echo "Cleaning up and restarting." | tee -a logs/sender.log
    rm -rf files/*.tar.gz

    if [ -f files/to_send/SUCCESS ]; then
        echo "Last transfer successful, preparing for next time" | tee -a
logs/sender.log
        rm -rf files/to_send/*
    else
        echo "Last transfer did not end correctly, preparing for retry" | tee -a
logs/sender.log
    fi
    date +"%T" | tee -a logs/sender.log
    ./bin/sender | tee -a logs/sender.log &
    echo "running ..." | tee -a logs/sender.log
  fi
done
```

O script acima verifica se uma instância do programa está rodando, caso negativo, loga o horário do reinício e a incia após realizar a limpeza dos arquivos temporários. A aplicação sender escreve um arquivo SUCCESS ao ser encerrada com sucesso na pasta files, de maneira que podemos checar por ele. Na hipótese de conclusão da transferência com sucesso, podemos apagar os arquivos da fila .

A aplicação unix tee serve para realizar a duplicação das mensagens, incluindo-as na saída padrão e nos arquivos contidos em logs/. Analogamente, o script run_forever_receiver.bash realiza o mesmo monitoramento do processo, entretanto este não realiza nenhuma limpeza ao final da execução, pois supõe que esta será feita no ato da recuperação dos arquivos pelo interessado.

Para transformar estas aplicações em um daemon linux que inicializa no boot, basta inserirmos a linha para sua execução no arquivo /etc/rc.local, ex:

```
su – USER –c /PATH/TO/BIN/run_forever_sender.bash &
```

Desta maneira, a aplicação é iniciada em segundo plano e sem um terminal acoplado, o que faz com que os dados sejam escritos apenas no arquivo de logs.

Teste de Campo

Após extensivos testes do protocolo em ambiente controlado (laboratórios), nos quais testamos a capacidade de transferência, tomamos o VANT e o RaspberryPi para um teste de campo. Este foi realizado no estacionamento da Faculdade de Tecnologia da Universidade Federal do Amazonas. O seu objetivo era realizar a prova do conceito, e garantir que a implementação se comportaria de maneira adequada no seu contexto de utilização.

Os dispositivos utilizados foram os seguintes:

- Um quadrotor ASCTEC Pelican com *dongle* WiFi USB
- Um Raspberry Pi 3 modelo B com adaptador WiFi integrado
- Um notebook com adaptador WiFi integrado

Todos os dispositivos foram configurados na mesma rede adhoc como descrito na seção Implementação, com endereços IPs distintos, de maneira que pudéssemos realizar acesso remoto nos sistemas finais utilizando o notebook via a aplicação ssh. Para este teste em especifíco, uma

configuração adicional foi necessária. Esta configuração trata de alterar os sockets nos dispositivos para permitir um TTL de no máximo 1 (UM). Ela é necessária para garantirmos que os pacotes transmitidos não transitem ao sistema final por meio do notebook, e foi removida da implementação final.

Os dispositivos (sensor e VANT) foram posicionados à uma distância de aproximadamente 170 metros um do outro, na qual notamos que os dispositivos não eram mais acessíveis entre si. Isso foi verfiicado utilizando-se a utilidade ping, que retornava o erro Destination Host Unreachable, o que indica que os dispositivos não estavam mais conectados na mesma rede.

Após isso, o notebook foi utilizado para iniciar as aplicações em ambos os sistemas finais em segundo plano. O VANT foi operado gradual e manualmente para mais perto do sensor, de maneira que à distância de aproximadamente 50 metros, o handshake do passo 8 do protocolo foi realizado com sucesso. Entretanto a transmissão confiável de dados só se inicializou de fato à distância de pouco mais de 20 metros.

Os dados foram transmitidos com sucesso com o drone em vôo, e atestamos a integridade dos dados recebidos comparando a saída da utilidade unix cksum. Pode-se verificar a execução com sucesso do protocolo pelo vídeo disponível em https://youtu.be/tYSsXEPDvz8.

Figura 7: Local de Teste do Protocolo

Conclusão

Neste trabalho detalhamos a concepção e implementação de um protocolo de transporte de dados utilizando VANTs para ser utilizado em aplicações onde o transporte de dados via rádio ou satélite não possui um alto custo/benefício.

Tal protocolo transporta dados arbitrários entre dois nós conectados em uma rede ad-hoc. Em nosso caso, um nó é um veículo aéreo não tripulado, e outro um Raspberry-Pi. Nossa solução pode ser facilmente generalizada para outras aplicações, visto que se restringe apenas as condições de existência de dois ou mais nós, conectados na mesma rede (adhoc ou não) e da API TCP/IP implementada nos sistemas finais.

Cronograma:

2017/01										
Descrição	14/04 - 21/04	22/04 - 28/04	29/04 - 05/05	06/05 - 12/05	13/05 - 19/05	20/05 - 26/05	27/05 - 02/06	03/06 - 09/06	10/06 - 16/06	17/06 - 26/06
Projeto	**X**	**X**								
Implementação		**X**	**X**	**X**						
Testes				**X**	**X**					
Adicionais					**X**	**X**	**X**			
Testes de Campo								**X**	**X**	**X**
Entrega										**X**