Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Curso de Graduação em Engenharia Elétrica

DENIS MARTINS DANTAS

DESENVOLVIMENTO DE UM DRIVER PARA MODEM GPRS BASEADO EM MODELOS

Campina Grande, Paraíba
Dezembro de 2015

DENIS MARTINS DANTAS

DESENVOLVIMENTO DE UM DRIVER PARA MODEM GPRS BASEADO EM MODELOS

Trabalho de Conclusão de Curso submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Controle e Automação

Orientador:
Professor Saulo Oliveira Dornellas Luiz

Campina Grande, Paraíba
Dezembro de 2015

DENIS MARTINS DANTAS

DESENVOLVIMENTO DE UM DRIVER PARA MODEM GPRS BASEADO EM MODELOS

Trabalho de Conclusão de Curso submetido à Unidade Acadêmica de Engenharia Elétrica da Universidade Federal de Campina Grande como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciências no Domínio da Engenharia Elétrica.

Área de Concentração: Controle e Automação

Aprovado em / /

Professor Avaliador
Universidade Federal de Campina Grande
Avaliador

Professor Saulo Oliveira Dornellas Luiz
Universidade Federal de Campina Grande
Orientador, UFCG

Esse trabalho é dedicado à minha família, que sempre me ajudou e guiou para que eu pudesse realizar os meus sonhos. É dedicado aos meus amigos de escola, que fizeram parte e ajudaram no meu crescimento pessoal. É dedicado aos meus amigos da universidade, com quem virei inúmeras noites estudando para provas (Arthur, Yuri, Ademir, Lucas, e João). Dedico também à minha namorada, Laryssa, que neste último ano me apoiou diante de todas as mudanças na minha carreira.

RESUMO

Este trabalho de conclusão de curso apresenta o desenvolvimento de um driver para um modem GPRS por meio do projeto baseado em modelos. As fases de desenvolvimento foram baseadas nos requisitos de um novo produto, com o objetivo de monitorar dados de veículos de uma frota, enviando informações em tempo-real para um servidor remoto. Este produto se posiciona num contexto que vem sendo chamado pelo mercado de Internet das Coisas (IoT). Uma arquitetura de *software* chamada Arquitetura de Três Camadas foi utilizada para definir vários módulos de *software* embarcado e como eles interagem entre si. O projeto realizado neste trabalho prevê a utilização de um Sistema Operacional em Tempo Real para dar suporte a um ambiente multitarefa, e para sincronização da operação dessas tarefas.

Palavras-chave: Desenvolvimento baseado em modelos, V-Model, Arquitetura de Software Embarcado, Sistema em tempo-real, Sistema Operacional em Tempo- Real, IoT.

ABSTRACT

This report presents the model-based development of a driver for a Modem GPRS, being used with a microcontroler. The development phases were based on the requirements of a new Internet of Things product, responsible for reading and sending vehicle related real-time data to a remote server. A Software Achitecture named Three-Layered Architecture (LRA) was used to define several software modules and their interactions. The design of the Project predicts the need to use a Real-Time Operating System to give support to multiple tasks running concurrently, and to provide sinchronization functionalities for a proper support of a multitask environment.

Keywords: Model Based Design, V-Model, Embedded Software Architecture, Real-Time Systems, Real-Time Operating System, IoT.

LISTA DE FIGURAS

GLOSSÁRIO

- OS: Do inglês, Sistema Operacional;
- RTOS: Do inglês, Sistema Operacional em Tempo Real;
- IoT: Do inglês, Internet das Coisas;
- M2M: Do inglês, [Comunicação] Máquina à Máquina;
- API: Do inglês, Interface de Programação de Aplicação;
- GPRS: Do inglês, Serviço de Rádio de Pacote Geral.

SUMÁRIO

1 INTRODUÇÃO

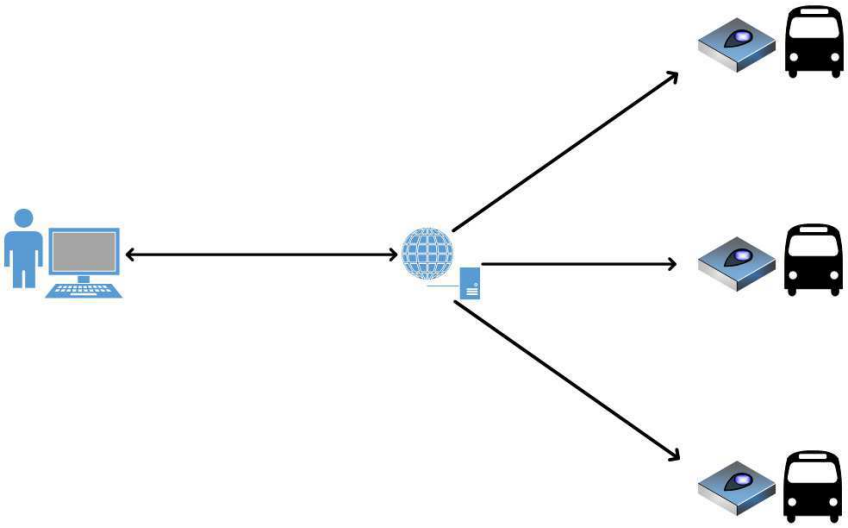
A utilização de dispositivos inteligentes conectados a redes (locais ou Internet) é uma tendência atual do mercado. O investimento em tecnologias de monitoramento remoto de objetos, carros, cidades e seus serviços, e comunicação máquina a máquina – uma esfera que vem sendo chamada de Internet das Coisas – veio aumentando progressivamente na última década (ver Fig. 2).

O foco do trabalho realizado neste projeto é a metodologia de desenvolvimento de um produto, partindo de especificações de requisitos do produto até chegar à fase de implementação. Portanto as fases iniciais descritas envolvem aspectos gerais do desenvolvimento de um dispositivo embarcado, enquanto a parte final é específica ao desenvolvimento de um driver que é utilizado no sistema.

A arquitetura externa de alto nível do dispositivo projetado é apresentada na Fig. 1. O dispositivo, instalado em veículos interage com um servidor web através de um Modem GPRS. Os dados no servidor são acessados por meio de um web site por um usuário do sistema.

O escopo da implementação realizada no projeto é o desenvolvimento de um driver para o modem GPRS. Este modem é normalmente utilizado em conjunto com algum microcontrolador como mestre. No entanto, o processo descrito neste trabalho é independente de plataforma, no que diz respeito às especificações do driver.

FIGURA 1 - ARQUITETURA EXTERNA DE ALTO NÍVEL.



2 REVISÃO

O contexto desse trabalho e das ferramentas a serem implementadas são apresentados neste capítulo.

2.1 INTERNET DAS COISAS (INTERNET OF THINGS - IOT)

(Vermesan & Friess, 2014) indicaram que:

> A Internet das Coisas (IoT) é a rede de objetos físicos que contem tecnologia embarcada [como microcontroladores e sensores] para se comunicar e medir seus estados internos e o ambiente externo [...]

Por objetos físicos pode-se entender como objetos, animais e pessoas.

Dentro da rede IoT, sistemas embarcados operam comunicando-se com um servidor, ou algum outro dispositivo remoto, para enviar dados de telemetria medidos do ambiente e/ou executar algum tipo de instrução (atuar). Os dispositivos embarcados adquirem dados de telemetria como temperatura, posição geográfica, ou pressão, através de sensores. A informação é enviada por uma estrutura de rede como TCP/IP, ou Zigbee.

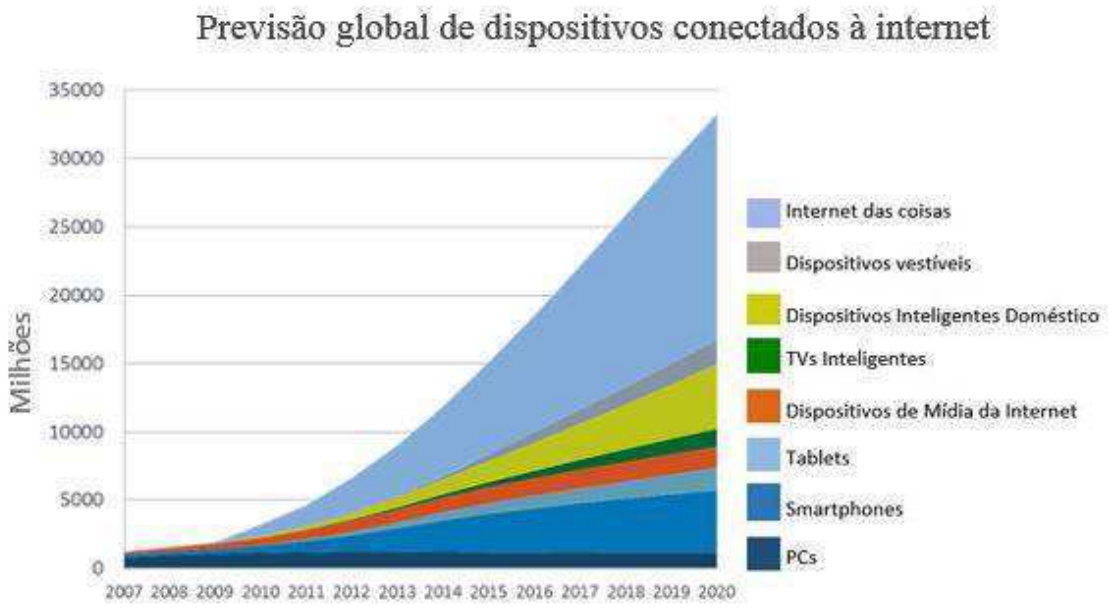
Um servidor remoto pode também enviar informações para um sistema embarcado contendo instruções para que esse dispositivo atue no ambiente – por exemplo, uma aplicação industrial que precisa ser controlada remotamente – ou por exemplo uma atualização dinâmica do próprio *firmware*[1], ou uma mudança de alguma configuração.

O crescimento estimado de dispositivos conectados à Internet, por categoria, incluindo dispositivos IoT, desde o ano de 2007 é apresentado na Fig. 2. De acordo com (Vermesan & Friess, 2014):

[1] Firmware: *Software* permanente programado em uma memória de *somente leitura*, tipicamente utilizado em sistemas embarcados.

> O número de dispositivos conectados à internet ultrapassou o número de seres humanos no planeta em 2011, e até 2020, é esperado que o número de dispositivos conectados à internet seja entre 26 e 50 bilhões.

FIGURA 2 - CRESCIMENTO DO NÚMERO DE DISPOSITIVOS CONECTADOS À INTERNET DESDE 2007 COM ESTIMATIVA ATÉ 2020



FONTE: HTTP://CONVERSATION.CIPR.CO.UK/2014/10/17/EXPLORE-INTERNET-THINGS-MARKET-OUTLOOK-2020/, ACESSADO EM 23/10/2015.

O desenvolvimento de dispositivos embarcados a serem inseridos no mercado de IoT necessita ser feito de forma eficaz no sentido de permitir a escalabilidade e mantenebilidade do sistema, de forma que uma empresa desenvolvedora possa absorver a demanda de mercado que é esperada. A necessidade de focar esforços em um sistema mantenível e escalável ocorre também devido às diversas áreas do mercado que irão absorver dispositivos inteligentes conectados à internet, fazendo com que a agilidade de uma empresa em desenvolver novas soluções seja fundamental.

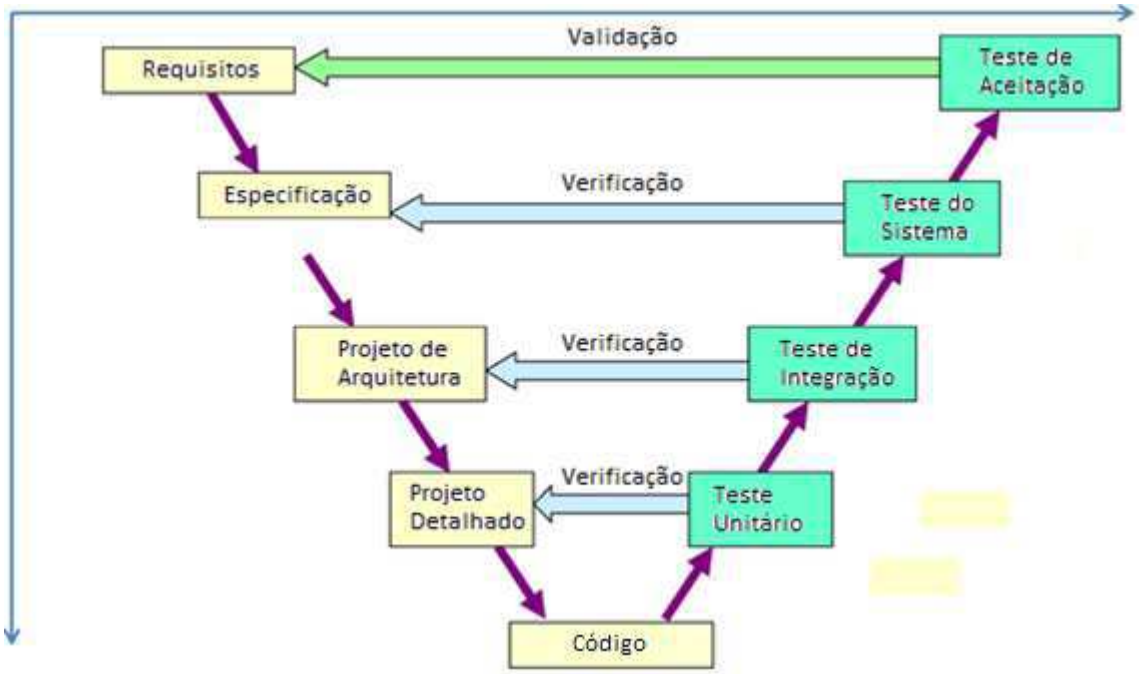
2.2 DESENVOLVIMENTO V-MODEL

O V-Model é um tipo de diagrama utilizado para desenvolvimento de sistemas que visa guiar as atividades que serão realizadas na fase no desenvolvimento, diminuindo assim a complexidade da execução. Uma clássica representação de um diagrama V-Model é apresentada na Fig. 3. As atividades mostradas no V-Model são agrupadas entre aquelas relacionadas ao desenvolvimento em si (parte esquerda do

diagrama), e as atividades relacionadas à Verificação e Validação (V&V – lado direito) do sistema. Como pode ser visto na Fig. 3, atividades de verificação e validação são aplicadas nas diversas fases realizadas na etapa de desenvolvimento.

Atividades de verificação são um conjunto objetivo de testes que confirmam a qualidade do produto como um todo, ou verifica se módulos individuais de *hardware* ou *software* alcançam as métricas determinadas nos requisitos. A validação busca demonstrar que o produto realiza sua intenção original.

FIGURA 3 - DIAGRAMA V-MODEL SIMPLIFICADO.



FONTE: FOWLER, KIM(2014). DEVELOPING AND MANAGING EMBEDDED SYSTEMS AND PRODUCTS: METHODS, TECHNIQUES, TOOLS, PROCESSES, AND TEAMWORK

Ao longo do processo de desenvolvimento, alguns documentos devem ser produzidos. Tais documentos são importantes para a futura validação e verificação do sistema, e para referência futura para desenvolvedores que possam eventualmente serem integrados à equipe de desenvolvedores.

Considerando o lado esquerdo da Fig. 3, o desenvolvimento começa definindo o ambiente do sistema e seus requisitos (funcionais e não-funcionais). A fase de *Especificação* irá definir como o sistema irá alcançar seus requisitos. A fase *Projeto de Arquitetura* define uma visão de alto nível sobre o *hardware* e *software* do sistema. A fase de *Projeto Detalhado* descreve em detalhes cada componente de *hardware* (pode-se fazer uma pesquisa de mercado nessa fase também), como eles se interconectam, segurança e robustez dos componentes. A Arquitetura de *Software* é detalhada com foco

em como cada módulo interage com os outros módulos e o ambiente. Finalmente, na fase *Código* a implementação começa – é possível começar a implementação de *software* e *hardware* em paralelo a partir da utilização de placas de desenvolvimento e construção de protótipos simplificados do sistema final.

Atividades V&V são realizadas do lado direito do V-Model. Testes unitários podem verificar se a implementação de pequenos módulos está correta testando entradas e saídas. Na *Integration Test* é feita a verificação da comunicação e interação entre os módulos. No *System Test* a verificação irá mostrar que o sistema funciona de acordo com os aspectos que foram definidos – como casos de uso, cenários e protocolos de comunicação. Finalmente, a *Acceptance Test* valida todos os requisitos de acordo com suas especificações.

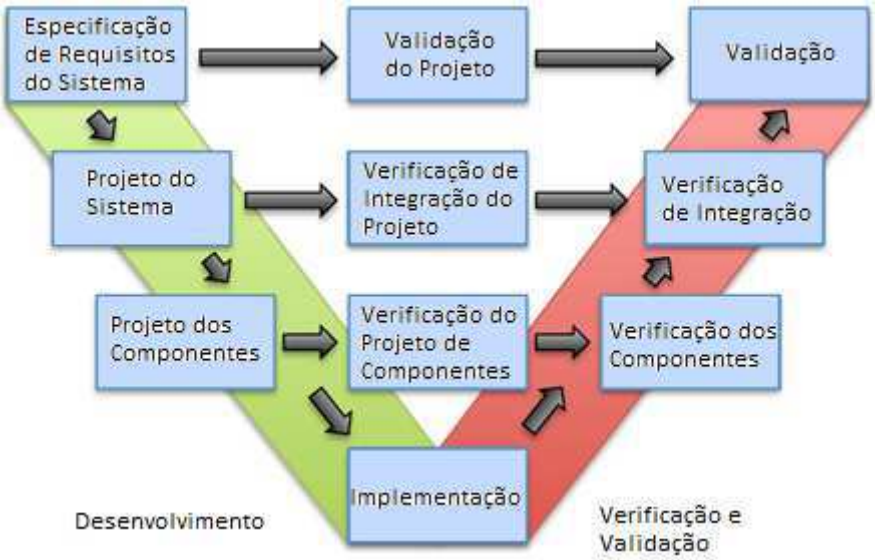
Idealmente uma equipe de testes deve ser atribuída para realizar as atividades V&V – uma equipe diferente é preferível por ser mais neutra em relação aos testes que irão executar, a avaliar de forma mais imparcial a mantenebilidade do sistema e se o sistema tem interface de usuário amigável ou não (*user-friendly*). Para facilitar o processor de Validação e Testes é importante ter documentos explicando os resultados esperados e os passos necessários para obter tais resultados.

Um outro diagrama V-Model levemente diferente é apresentado na Fig. 4. Este diagrama especifica fases de atividades de planejamento – por exemplo, um plano de validação é produzido logo depois que a fase *System Requirements Specification* é realizada. A atividade de validação em si é realizada do lado direito na *Integration Verification.*

(Scippacercola, Pietrantuono, Russo, & Zentai, 2015) explica:

> O desenvolvimento como definindo o ambiente do sistema e seus requisitos (funcionais e não-funcionais). Depois, [as fases] *Projeto do Sistema* e *Projeto dos Componentes* são realizadas. A primeira define uma arquitetura de alto nível do sistema e distingue as partes que devem ser realizadas por *hardware* daquelas que devem ser implementadas por *software*. Os requisitos são então associados a componentes, e novos requisitos podem ser descobertos para apontar interações apropriadas entre os elementos. A fase *Projeto dos Componentes* define a arquitetura interna de cada componente. Finalmente, o lado de desenvolvimento do diagrama procede com a Implementação.

FIGURA 4 - MODELO INCLUINDO ETAPAS DE PLANEJAMENTO DA VERIFICAÇÃO E VALIDAÇÃO.



FONTE: PIETRANTUONO, ROBERTO (2015). MODEL-DRIVEN ENGINEERING OF A RAILWAY INTERLOCKING SYSTEM.

2.3 PROJETO BASEADO EM MODELO

"No projeto e desenvolvimento de *software* baseado em modelo, a modelagem de *software* é utilizada como uma parte essencial do processo de desenvolvimento do *software*" (Gomma, 2011). O comportamento do sistema a ser implementado é explicado através de modelos definidos em linguagem gráfica UML – SysML se estende para definir sistemas em geral. Modelos são construídos e analisados para prover aos desenvolvedores um caminho guiado para cumprir os requisitos do projeto.

De acordo com (Gomma, 2011):

> O objetivo do desenvolvimento baseado em modelo é utilizar modelos como artefatos principais no desenvolvimento em todas as fases do processo de desenvolvimento. Isto promete aumentar a produtividade e qualidade do processo de desenvolvimento de *software* através do aumento do nível de abstração que é feito no desenvolvimento, assim como no grau de automação, com a ajuda de modelos que são adaptados e apropriados para o tipo específico de tarefas a serem desenvolvidas.

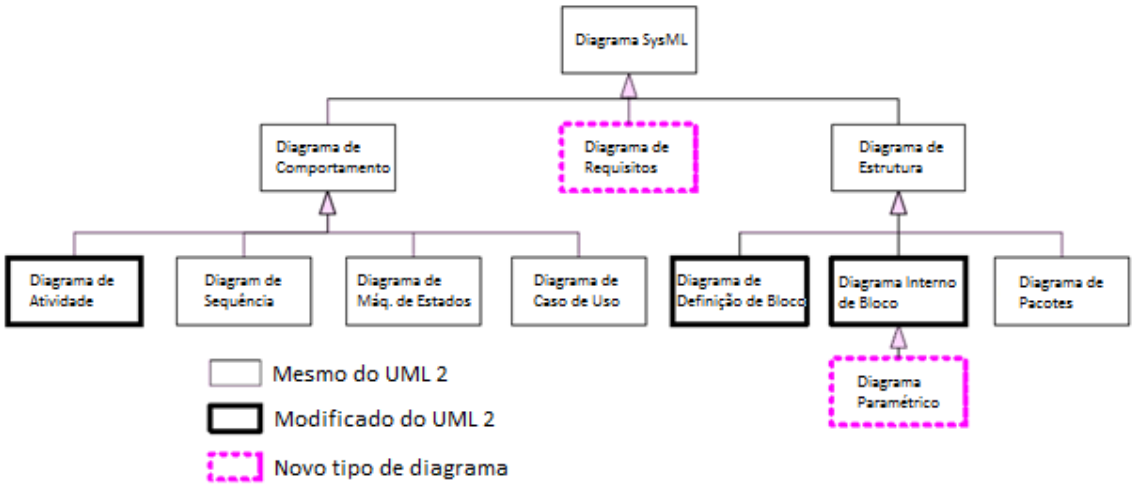
SysML adotou alguns diagramas UML, modificou alguns do já existentes e criou dois outros. Os diagramas podem ser divididos pelo que pretendem representar: Comportamento, Requisitos e Estrutura. Um gráfico hierárquico mostra a relação dos diagramas de SysML na Fig. 5.

- Diagrama de Comportamento – composto por:
  - Diagrama de Atividade (modificado): O diagrama de atividade é utilizado para especificar o fluxo de entradas, saídas, e controle, incluindo sequência e condições para atividades coordenadas. Em SysML, diagramas de atividade suportam modelagem de fluxo contínuo;
  - Diagrama de Sequência: Utilizado para modelar o fluxo da lógica de um sistema de forma visual, permitindo tanto documentação quanto validação dessa lógica. Esse diagrama é comumente utilizado com propósito de análise e projeto. Ele foca em identificar o comportamento de um sistema a partir de interações entre partes do sistema;
  - Diagrama de Máquina de Estado: Utilizado para representar o "ciclo de vida" de um bloco em relação às suas ações no sistema. Ele dá suporte a um comportamento baseado em eventos. Cada estado tem uma atividade *entry* (ação a ser realizada quando se acaba de entrar em um estado)*, exit* (ação a ser realizada quando se sai do estado) e *do* (ação a ser realizada a cada ciclo dentro do estado, sendo a ação realizada novamente caso não haja uma transição de estado). Transições modificam o estado atual de acordo com condições predefinidas;
  - Diagrama de Caso de Uso: Provê meios de descrever funcionalidades básicas em temos de uso do sistema e objetivos – isto é feito utilizando atores (partes externas que agem sobre o sistema). Funcionalidades comuns podem ser fatoradas através de relações *include* e *extend*.
- Diagrama de Requisitos: Estes diagramas são compostos por estereótipos *requisitos*, que representam descrições de requisitos baseadas em texto e identificadas com um ID único. O diagrama cria uma hierarquia de requisitos pertencentes a alguma especificação;
- Diagrama de Estrutura – composto por:
  - Diagrama de Definição de Bloco (*Block Definition Diagram – BDD,* modificado): Blocos são definições UML de classes. Ele descreve um conceito unificado para explicar a estrutura de um elemento ou sistema

(*hardware*, *software*, dado, etc). Múltiplos compartimentos descrevem graficamente as características do bloco (propriedades, operações, limitações, etc.). O BDD tem objetivo de mostrar as relações entre diferentes partes do sistema – cada uma dessas partes sendo um sistema menor.

- Diagrama Interno de Bloco (*Internal Block Diagram* – IBD, *modificado*): Tem o objetivo de mostrar a composição interna de um bloco. Geralmente é utilizado para especificar o interior de um bloco que foi previamente definido em um contexto geral pelo BDD;
- Diagrama de Pacotes: Utilizado para organizar o modelo. Agrupa elementos de modelo em um único *namespace*. Pode especificar o uso de elementos, limitações e regras de filtragem;
- Diagrama Paramétrico: Utilizado para expressar relações (na forma de equações) entre valores e propriedades de valores – dá suporte para análise de engenharia (desempenho, confiabilidade, etc). Um *bloco de relação* captura equações, e um Diagrama Paramétrico representa o uso de relações em um contexto de análise.

FIGURA 5 - DIAGRAMAS SYSML.

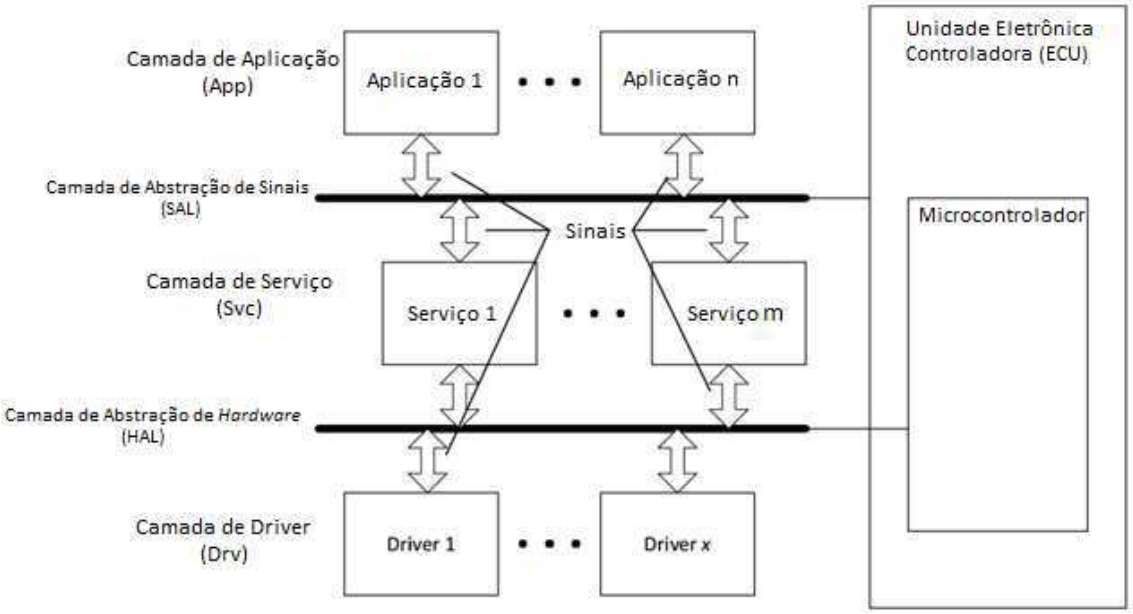


FONTE: (FRIEDENTHAL, MOORE, & STEINER, 2015)

2.4 ARQUITETURA DE TRÊS CAMADAS (THREE-LAYERED ARCHITECTURE)

A Arquitetura de Três Camadas é uma arquitetura de *software* inspirada no AUTOSAR[2]. Essa arquitetura define três camadas de módulos de *software*: Camada de Aplicação, Camada de Serviço e Camada de Driver. Na Fig. 6 é apresentado um diagrama com os módulos e interfaces da arquitetura.

FIGURA 6 – ARQUITETURA DE SOFTWARE DE TRÊS CAMADAS.



FONTE: (BUENO, MORENO, & SWEET, 2013)

"A Camada de Driver contem comunicações de baixo nível, controles, comandos e procedimentos. Essa camada é a que manipula os registradores e pinos do microcontrolador diretamente, e interage com o *hardware* e o ambiente externo" (Bueno, Moreno, & Sweet, 2013). A Camada de Driver é uma abstração do *hardware* subjacente – GPIOs, memória interna, dispositivos externos, etc. Os módulos de driver contem APIs[3] que são fornecidas para outros módulos de driver e serviço utilizarem. Os módulos de driver também podem implementar tarefas que são executadas indefinidamente.

[2] AUTOSAR: http://www.autosar.org/
[3] Application Program Interface (Interface de Programação de Aplicação, em português).

A Camada de Serviço contém módulos de *software* responsáveis por controlar a funcionalidade dos Drivers, e de converter dados para uma forma que possa ser diretamente utilizada pelos módulos da camada de aplicação. Módulos de Serviço podem ser compostos por APIs (utilizadas por outros Serviços e Aplicações) e podem também implementar tarefas rodando máquinas de estado, por exemplo.

A Camada de Aplicação contém módulos que irão implementar as funcionalidades principais observadas pelo usuário. O projeto dos módulos de Aplicação é baseado nos requisitos funcionais do produto. Cada módulo de Aplicação deve ser implementado por uma máquina de estados.

Se o *hardware* for modificado por um modelo diferente, a Camada de Serviço deverá ter o mínimo de modificações possíveis, e a camada de aplicação deverá ser intocada. Ao mesmo tempo, essa arquitetura é flexível para portabilidade entre diferentes projetos utilizando o mesmo *hardware*. A camada de aplicação é a única modificada em projetos com requisitos funcionais diferente que podem ser implementados no mesmo *hardware*.

O projeto dessa arquitetura é orientado à implementação de *software* embarcado. Fazendo uma analogia a uma arquitetura de sistema operacional de propósito geral, os Drivers e Serviços seriam o kernel, enquanto a Aplicação são os programas do usuário.

Um Sistema Operacional é previsto para fornecer serviços de escalonamento das tarefas em cada um dos módulos, além de possivelmente oferecer outros serviços que venham a facilitar a execução apropriada dos módulos de *software*.

2.5 SISTEMAS OPERACIONAIS EM TEMPO-REAL

Um sistema em tempo-real é um sistema que que reage a estímulos externos realizando ações baseadas em tais eventos – a ação, ou resposta, é dada dentro de um certo período de tempo máximo pré-definido. Para o resultado da operação ser correto é preciso não só o resultado da operação estar correto, mas também ter sido realizada dentro do prazo limite estipulado.

Sistemas em tempo-real podem ser implementados através da execução de uma ou mais *tarefas*, que "[...] é um programa sequencial que realiza uma certa computação e possivelmente se comunica com outras tarefas em um sistema". (Isovic, 2012). Quando múltiplas tarefas devem executar para efetuar um (ou mais) objetivos de um

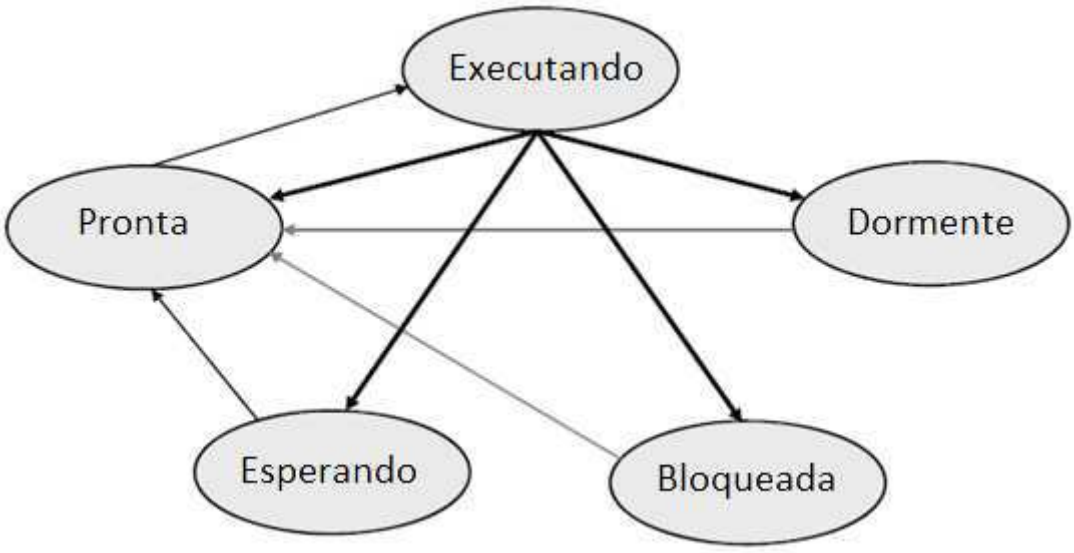
sistema em tempo-real, tais tarefas irão concorrer pelos recursos do sistema. Em um sistema de processador único é necessário compartilhar o tempo de processamento entre as tarefas de uma forma que simula a execução paralela. O motivo pelo qual isto é uma simulação de paralelismo é que um processador só é capaz de executar instruções (um de cada vez) de cada tarefa de forma serial.

A principal responsabilidade de um Sistema Operacional (SO) é escalonar a execução de várias tarefas durante seu tempo de atividade. "Um Sistema Operacional é um programa que controla a execução de aplicações e atua como uma interface entre as aplicações e o *hardware* do computador" (Stallings, 2008). Um Sistema Operacional em Tempo-Real (*Real-time Operating System – RTOS)* suporta a execução de aplicações de tempo-real (ou tarefas).

2.5.1 ESTADOS DE TAREFAS

Tarefas podem modificar entre diferentes estados durante sua execução. Um modelo de dois estados define tarefas entre *rodando* ou *não-rodando*. No entanto, típicos sistemas em tempo-real dividem o estado *não-rodando* em quatro outros estados: *dormente (dormente), pronta (ready), esperando (waiting)* e *bloqueada (blocked)*. O modelo de cinco estados é representado na Fig. 7, onde rodando se torna *executando* (running).

FIGURA 7 – ESTADOS DE UMA TAREFA E AS POSSÍVEIS TRANSIÇÕES.



FONTE: (ISOVIC, 2012)

- Dormente: Este estado significa que a tarefa ainda não está consumindo nenhum recurso do sistema. A tarefa está registrada no sistema, mas ainda não está ativada ou terminou sua execução;
- Executando: Quando a tarefa entra nesse estado, suas instruções começam a serem executadas no processador. Apenas uma tarefa por vez pode ser executada (em um sistema com um único núcleo de processamento);
- Pronta: Este estado sinaliza que a tarefa está pronta para ser executada pelo processador. Uma tarefa pronta não pode ganhar o controle da CPU até que todas as tarefas com prioridades maiores com estado *pronta,* ou *executando*, completem ou se tornem dormentes;
- Esperando: Uma tarefa entre nesse estado quando ela está esperando por um evento (e.g. *timeout* expirado), ou um sinal de sincronização, de outra tarefa. Uma função *sleep* coloca a tarefa neste estado.
- Bloqueada: Uma tarefa é bloqueada quando ela foi colocada para executar mas não pode continuar porque sua execução foi bloqueada por alguma razão. Por exemplo, ela pode estar bloqueada esperando um recurso compartilhado ser liberado.

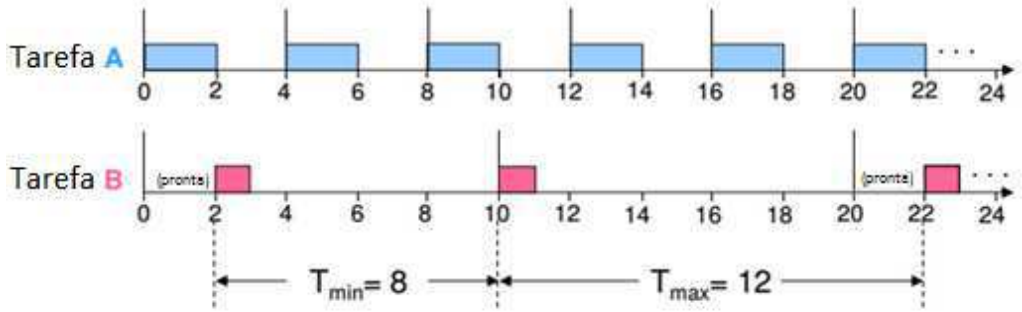
O estado *esperando* é comumente utilizado para implementar tarefas periódicas com um RTOS. Uma tarefa periódica executa o mesmo escopo de instruções periodicamente. Uma vez que o escopo termina ela chama uma função *sleep*, fornecida pelo RTOS, que aceita um período de tempo como argumento, indicando o quanto esta tarefa deve permanecer no estado *esperando*.

2.5.2 PRIORIZAÇÃO E ESCALONAMENTO

Um outro mecanismo implementado por um RTOS é a priorização de tarefas. Algumas tarefas podem ser mais críticas em relação a executarem no momento certo do que outras. Por essa razão um RTOS associa prioridades às tarefas. As prioridades são escolhidas pelo desenvolvedor quando as tarefas são criadas – alguns sistemas também oferecem mecanismos de mudança dinâmica de prioridades. Se uma tarefa de alta prioridade fica pronta para executar quando uma tarefa de baixa prioridade está executando, então a primeira tarefa irá interromper a execução da última, i.e., o RTOS irá direcionar a CPU para executar a rotina da tarefa de maior prioridade.

O agendamento da execução de tarefas (ou a escolha de qual tarefa irá executar em cada momento) é feito de forma dinâmica, em tempo de execução, pelo *escalonador* do RTOS – este processo de agendamento é chamado de escalonamento. Um escalonamento simples é demonstrado pelo plano de execução de duas tarefas periódicas na Fig. 8.

FIGURA 8 – ESCALONAMENTO DE DUAS TAREFAS COM DIFERENTES PRIORIDADES.



FONTE: MÄLARDALEN UNIVERSITY, DISCIPLINA EMBEDDED SYSTEMS 2.

A tarefa A tem tempo de execução igual a 2 unidades de tempo e período igual a 4 unidades de tempo, enquanto a tarefa B tem tempo de execução igual a 1 unidade de tempo e período igual a 10 unidades de tempo. No instante 0 é considerado o pior cenário, pois as duas tarefas estão prontas para executar simultaneamente. Como a tarefa A tem maior prioridade o RTOS coloca ela para executar, e a tarefa B passar a ficar esperando. No instante 2, a tarefa B executa, e termina sua execução no instante 3. No instante 10, a tarefa B está pronta para executar novamente, mas dessa vez o processador está livre, portanto a tarefa B pode executar. No instante 20 as duas tarefas estão prontas para executar simultaneamente, como em 0. Portanto o intervalo de tempo igual a 20 unidades é considerado um *hiperperíodo*, i.e., o período em que o plano de execução de tarefas se repete.

2.5.3 SINCRONIZAÇÃO DE TAREFAS

Frequentemente existem tarefas que precisam executar assincronamente, mas podem precisar interagir umas com as outras, por exemplo para enviar dados uma para as outras, ou para acessar um recurso local (e.g. memória compartilhada ou *hardware* de comunicação). A maioria dos RTOSs oferecem uma variedade de mecanismos para manipular as interações entre tarefas. Operações realizadas com recursos compartilhados são consideradas *sessões críticas,* e precisam ser *mutuamente exclusivas*

para evitar colisões danosas (de execução) entre tarefas que também se utilizam do recurso compartilhado. Ou seja, durante as sessões críticas, uma tarefa não pode ser interrompida para que outra tarefa manipule o mesmo recurso compartilhado que está sendo manipulado pela primeira.

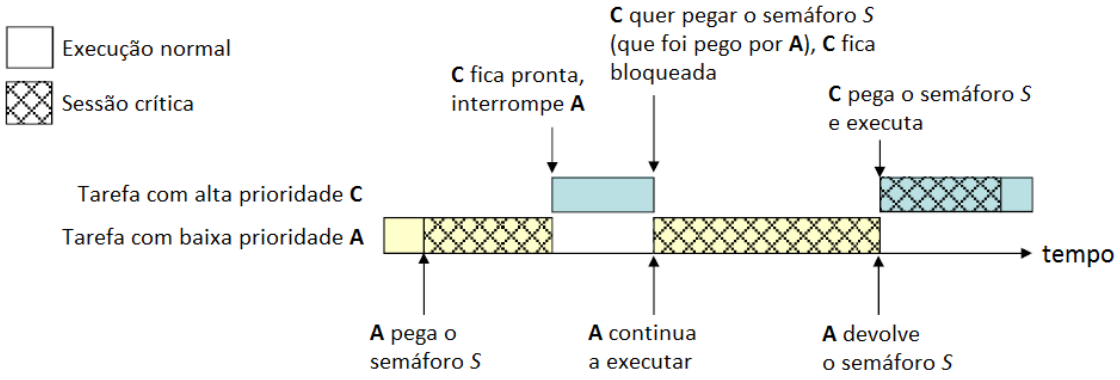
Uma solução possível para evitar interrupções durante uma sessão crítica é desabilitar as interrupções do sistema durante a operação da sessão crítica. No entanto esta abordagem pode ser muito danosa ao sistema como um todo, porque tarefas de alta prioridade que não utilizam o recurso compartilhado também poderão ser interrompidas.

Um *Semáforo Mutex* é uma ferramenta de *software* utilizada para alcançar a exclusão mútua de operações, ou sincronização do uso de um recurso compartilhado entre tarefas. A maior vantagem do semáforo sobre a estratégia de desabilitar interrupções é que ele não bloqueia o sistema inteiro. Semáforos possuem duas operações: *pegar* e *dar.*

Uma tarefa chama o método *pegar* para tentar "trancar" um semáforo. Se o semáforo estiver livre, a tarefa irá executar sua sessão crítica e posteriormente irá chamar o método *dar* para liberar o semáforo. Se uma tarefa tenta *pegar* um semáforo que está trancado, o estado desta tarefa passa a ser *bloqueado* até que o semáforo seja liberado novamente. Semáforos Mutex só podem ser liberados pela mesma tarefa que o trancou.

Um plano de execução de duas tarefas, com prioridades diferentes, utilizando um semáforo para sincronização de um recurso compartilhado é apresentado na Fig. 9.

FIGURA 9 – PLANO DE EXECUÇÃO DE DUAS TAREFAS UTILIZANDO UM SEMÁFORO.



FONTE: MÄLARDALEN UNIVERSITY, EMBEDDED SYSTEMS 2 COURSE, LECTURE 8.

A API *pegar*, fornecida pelo RTOS, normalmente irá receber como argumento uma identificação (ID ou variável do tipo semáforo) do semáforo e um *timeout*

indicando o tempo máximo que a tarefa deverá ficar bloqueada esperando caso o semáforo não esteja disponível. No momento em que *timeout* expira, a API deve retomar a execução retornando um indicativo que a tentativa de pegar o semáforo foi falha.

2.5.4 COMUNICAÇÃO DE TAREFAS

Comumente em um sistema multitarefa, i.e., onde várias tarefas executam concorrentemente, as tarefas precisam comunicar-se entre si. A maneira mais simples de realizar a comunicação entre tarefas é utilizando uma variável global compartilhada (ou um buffer). (Isovic, 2012) explica que:

> Comunicação através de memória compartilhada é um jeito simples e eficiente de [implementar] comunicação. [Esta abordagem] oferece um meio de comunicação inter-tarefa de baixo-nível, alta banda passante e baixa latência. É comumente utilizada para comunicação entre tarefas que rodam em um processador, assim como em tarefas que rodam em sistemas com processadores fortemente acoplados. [...] A desvantagem com essa abordagem é a sobrescrita de dados, i.e., valores antigos são sobrescritos por novos valores já que um buffer não é disponibilizado. Outra dificuldade é a sincronização do acesso à memória compartilhada. O desenvolvedor da aplicação precisa ter certeza que o acesso ao dado é feito de forma atômica, i.e., a tarefa não pode ser interrompida enquanto estiver atualizando o espaço de memória compartilhada. Isso pode ser alcançado por proteção da variável com um semáforo.

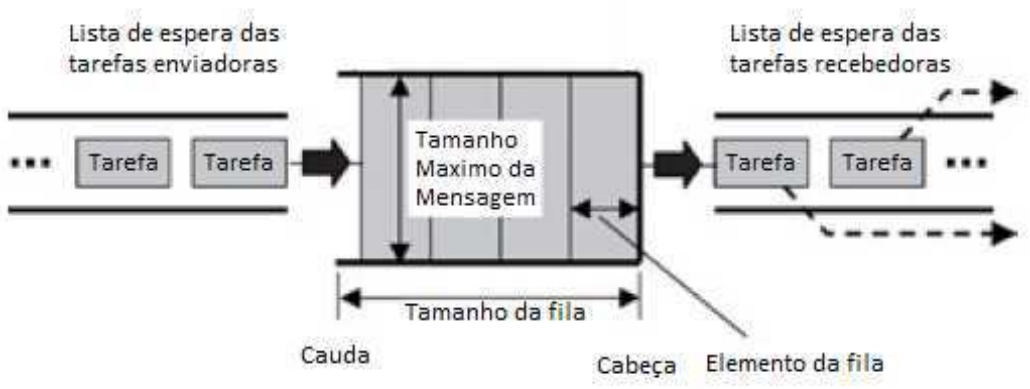
Mensagens passantes é a técnica mais popular para transferência de dados entre tarefas em um ambiente multitarefa. A maioria dos RTOSs utilizam mensagens passantes indiretas. Por essa abordagem as mensagens não são enviadas diretamente de uma tarefa para outras, mas através de *filas de mensagens*, que podem ser acessadas por diferentes tarefas.

De acordo com (Li & Yao, 2003), "Uma Fila de Mensagem é um objeto similar a um buffer, em que tarefas e ISRs[4] enviam e recebem mensagens para comunicar e sincronizar dados". O número máximo de mensagens que podem ser armazenados em uma fila é chamado de *capacidade* da fila, onde cada mensagem pode ser uma variável,

[4] Interrup Service Routines (do inglês, Rotinas de Serviço de Interrupção).

ou objeto (programação orientada a objeto). A ideia é que uma tarefa irá enviar uma mensagem para a fila, e eventualmente outra tarefa poderá retirar esta mensagem da fila (realizar uma leitura). Na Fig. 10 apresenta-se uma representação de uma fila.

FIGURA 10 – REPRESENTAÇÃO DE UMA FILA.



FONTE: (LI & YAO, 2003)

É dever do RTOS associar um ID único à fila. O desenvolvedor cria uma fila especificando parâmetros como sua largura (tamanho de cada objeto ou variável da fila) e a capacidade da fila. Uma vez que o RTOS tem esses dados, ele pode alocar espaço na memória para a fila.

A implementação de recebimento de dados de uma fila pode ser *bloqueante* ou *não-bloqueante*: a tarefa que quer fazer a leitura pode ficar bloqueada esperando uma nova mensagem chegar em uma fila que esteja vazia (é dever do desenvolvedor estabelecer um *timeout* para este tempo de bloqueio). Por outro lado, implementações de envio de dados para uma fila geralmente são *não-bloqueantes*: a tarefa que envia continua sua execução caso exista ou não uma tarefa esperando por uma mensagem da fila.

3 DESENVOLVIMENTO

Este capítulo descreve as fases de desenvolvimento de um dispositivo embarcado que realiza o monitoramento de informações de um veículo. Os dados colhidos são enviados em tempo-real para um Servidor *Web*, responsável por apresentar os dados em um *website.*

O desenvolvimento discutido neste trabalho é guiado pelas fases do diagrama V-Model descrito na Fig. 3. As fases podem ser adaptadas de acordo com os documentos que foram considerados apropriados ter como referência futura para o projeto. Este trabalho aborda as seguintes fases relacionadas ao desenvolvimento de um dispositivo embarcado:

1. Especificação de Requisitos do Sistema;
2. Projeto do Sistema;
3. Projeto dos Componentes;

3.1 ESPECIFICAÇÃO DE REQUISITOS DO SISTEMA

Essa fase começa com uma reunião envolvendo membros relacionados ao sistema, ou dispositivo, a ser desenvolvido. O encontro tem como intuito fazer um levantamento de requisitos, que envolvem, mas não necessariamente se limitam a:

- Definição sobre o que o dispositivo, e outras partes do sistema, precisam realizar para alcançar os requisitos do cliente;
- O número de dispositivos que deverão ser produzidos e seu valor teto de mercado;
- O ambiente em que o dispositivo embarcado irá operar: condições climáticas, estresse físico ou elétrico que o sistema deve suportar, e potencial tentativa de danificar o sistema;
- Definição de quais profissionais irão interagir com o dispositivo: quem irá instalá-lo, quem será o usuário, quem irá realizar manutenção e de que forma ocorre interação com o sistema;

- Definição de protocolo de comunicação: Caso haja comunicação com outros sistemas, deve-se definir um protocolo para estes sistemas comunicarem entre si.

A reunião deve ser gravada e devem ser feitas anotações sobre os tópicos discutidos. Depois, alguns membros envolvidos devem escutar o áudio do encontro e refinar suas anotações.

3.1.1 DECLARAÇÃO DO PROBLEMA

A Empresa Cliente possui uma frota de caminhões que são encarregados de realizar entregas em diversas cidades do Nordeste. As estradas por onde os caminhões trafegam possuem qualidade variada em relação aos buracos no asfalto. É sabido que certas peças se danificam mais rápido se o veículo for exposto a buracos, e quanto maior a velocidade mais rápida a deterioração dessas peças.

A realização de manutenção em caminhões da frota da Empresa Cliente é realizada atualmente de forma periódica, afim de identificar possíveis necessidades de troca de peças. A realização da manutenção preventiva é uma tarefa custosa para a empresa, e leva em consideração apenas o tempo entre uma revisão e outra.

Além da manutenção das peças, a empresa cliente notou que a realização da troca de óleo dos seus veículos vem sendo frequentemente realizada após a quilometragem limite estabelecida. À medida que a frota da empresa aumentou, foi ficando difícil o acompanhamento diário da quilometragem rodada de cada veículo da frota.

A empresa cliente procura uma solução que otimize a sua realização de manutenções veicular, de forma que a troca de óleo seja realizada sempre no momento correto, e que a manutenção preventiva seja feita dando-se prioridade a veículos da frota que tenham mais probabilidade de apresentarem problemas (baseado no estresse físico ao qual cada veículo foi exposto, velocidades durante os trajetos e quilômetros rodados).

3.1.2 DECLARAÇÃO DO PRODUTO

Para a Empresa Cliente o produto tem a função de monitorar em tempo-real informações relacionadas ao veículos de sua frota que auxiliem na eficiência da

manutenção preventiva e troca de óleo dos veículos. O produto deverá coletar informações em tempo-real, como: quilometragem rodada de cada veículo, velocidades instantâneas, e taxa e intensidade de trepidações.

As informações coletadas pelo produto deverão ser enviadas para um site, desenvolvido pela Empresa Desenvolvedora. Através deste site, a Empresa Cliente poderá ver informações dos veículos da frota da empresa e as informações pessoais dos condutores do veículo. Para cada veículo a empresa poderá ver as informações coletadas pelo produto, um histórico com data e horário das ocorrências (gráficos com velocidade instantânea e trepidações ao longo de cada dia).

O produto deve, ainda, ser de fácil instalação. A instalação do sistema nos veículos será feita pelo Técnico de Manutenção da Empresa Cliente. O sistema não pode de qualquer forma atrapalhar o motorista durante a condução do veículo.

Para a Empresa Desenvolvedora, o produto deverá ser de fácil manutenção.

3.1.3 Definição dos Usuários

Os seguintes usuários interagem com o dispositivo embarcado de forma direta ou indireta.

3.1.3.1 Usuário Leitor

- Descrição: O Usuário Leitor é um usuário da Empresa Cliente que faz a leitura das informações coletadas pelo equipamento em cada veículo.
- Ambiente do usuário: Este usuário acessa as informações através do *web* site fornecido pela Empresa Desenvolvedora.

3.1.3.2 Usuário Motorista

- Descrição: O Usuário Motorista conduz o veículo. Este usuário não tem nenhum conhecimento sobre a operação interna do dispositivo, apenas será notificado por aviso sonoro caso o equipamento apresente alguma falha.
- Ambiente do usuário: O Usuário Motorista irá conduzir o veículo (junto com o sistema instalado) em estradas BR do Nordeste. O veículo normalmente circulará em regiões com altas temperaturas e forte exposição a estresse físico.

3.1.3.3 USUÁRIO INSTALADOR

- Descrição: O Usuário Instalador é responsável por instalar o dispositivo em um veículo. Este usuário é um técnico contratado pela Empresa Cliente, que tem conhecimentos básicos de eletrônica. O Usuário Instalador realiza a ativação do sistema no veículo – e verifica seu funcionamento correto.
- Ambiente do usuário: O Usuário Instalador pode trabalhar tanto na sede da Empresa Cliente, quanto de forma móvel, indo até diferentes locais onde se encontrem os veículos.

3.1.3.4 USUÁRIO MANTENEDOR

- Descrição: O Usuário Mantenedor é um empregado Engenheiro da Empresa Desenvolvedora. Este usuário tem conhecimento do *hardware* e *firmware* do sistema instalado nos veículos. Este usuário é responsável por concertar equipamentos defeituosos.
- Ambiente do usuário: Este usuário trabalha na sede da Empresa Desenvolvedora, e tem acesso a equipamentos de manuseio e depuração da parte eletrônica, assim como equipamentos de depuração do *firmware*.

3.1.4 REQUISITOS

Esta sessão descreve os requisitos que são relevantes ao desenvolvimento do driver do sistema. Os requisitos estão divididos em relação aos usuários aos quais os requisitos são relevantes. Cada requisito é indicado pelo sigla FR, que significa *Requisito Funcional* (do inglês, *Functional Requirement)*.

É importante perceber que o produto, ou serviço, como um todo possui outros requisitos que não serão listados aqui porque o foco deste trabalho é no desenvolvimento de um Dispositivo Embarcado, e no driver de um modem.

Nesta sessão foram consideradas duas entidades no sistema projetado: um Dispositivo Embarcado (sistema embarcado que será instalado em veículos) e um Servidor Web.

3.1.4.1 REQUISITOS DO USUÁRIO LEITOR

FR1. Visualizar informações de caminhões em tempo-real.

- Componentes Relacionados: Dispositivo Embarcado e Servidor Web;
- Prioridade: Alta;
- Envolvidos: Usuário Leitor;
- Descrição: É de interesse dos usuários envolvidos observar, via Servidor Web, detalhes e informações sobre os veículos da frota: velocidade instantânea, quilometragem rodada e trepidações às quais o veículo foi exposto.
- Detalhes: A velocidade instantânea deve ser a velocidade média em um pequeno período de tempo de amostragem – o mesmo deve ser considerado para trepidações. As informações mostradas em tempo-real devem ser disponibilizadas em um gráfico mostrando os dados instantâneos da última hora (em relação à hora atual). A quilometragem deve ser atualizada em tempo-real, mas com período de amostragem de uma hora.

FR2. Visualizar histórico de informações coletadas dos veículos

- Componentes Relacionados: Dispositivo Embarcado e Servidor Web;
- Prioridade: Alta;
- Envolvidos: Usuário Leitor;
- Descrição: É de interesse dos usuários envolvidos visualizar o histórico dos dados coletados de cada veículo. O histórico deve permitir visualização por dia e por intervalo de tempo escolhido;
- Detalhes: Os usuários podem selecionar e salvar parâmetros limites que ficam marcados por uma linha horizontal nos gráficos. Caso as medições excedam o limite, os gráficos devem destacar em vermelho as medições excedentes.

FR3. Visualizar lista de veículos que devem passar por manutenção.

- Componentes Relacionados: Dispositivo Embarcado e Servidor;
- Prioridade: Alta;
- Envolvidos: Usuário Leitor;
- Descrição: É de interesse dos usuários envolvidos verificar com antecedência quais veículos precisarão fazer manutenção no próximo mês.
- Existem dois tipos de manutenção: preventiva e corretiva. A manutenção preventiva se caracteriza pela verificação de peças que podem se

desgastar ao longo do tempo – deve ser feita baseada na quilometragem rodada e na trepidação (estresse no chassi) pela qual passou o veículo. A manutenção corretiva relaciona-se a troca de componentes: óleo lubrificante e pneus. Para cada veículo deve-se ter parâmetros indicando o momento de realizar cada tipo de manutenção.

3.1.4.2 REQUISITOS DO USUÁRIO MOTORISTA

FR4. Verificar se o Dispositivo Embarcado está funcionando corretamente.

- Componentes Relacionados: Dispositivo Embarcado;
- Prioridade: Média;
- Envolvidos: Usuário Motorista, Usuário Instalador e Usuário Mantenedor;
- Descrição: O usuário envolvido deve ser capaz de saber se o Dispositivo Embarco do veículo apresenta problemas operacionais. A interação entre o Dispositivo Embarcado e o usuário envolvido ocorre por sinalização sonora vindo do Dispositivo Embarcado.
- Detalhe: O Dispositivo Embarco deve ser ligado no momento da partida do veículo. Quando o Dispositivo Embarcado ligar deve ser realizada uma auto-verificação, e caso o Dispositivo Embarcado encontre alguma falha interna, deve ser acionado o Indicador Sonoro (5 indicações sonoras em 15 segundos) para que o usuário envolvido saiba que o Dispositivo Embarcado não está com sua funcionalidade comprometida.

3.1.4.3 REQUISITOS DO USUÁRIO INSTALADOR

O requisito FR4 é comum ao Usuário Instalador.

FR5. Instalar o Dispositivo Embarcado na bateria do veículo.

- Componentes relacionados: Dispositivo Embarcado;
- Prioridade: Alta;
- Envolvidos: Usuário Instalador e Usuário Mantenedor;
- Descrição: O usuário envolvido deve ser capaz de instalar o Dispositivo Embarcado ligando-o na bateria do veículo.
- Detalhes: A bateria do veículo deve ser a fonte única de energia do Dispositivo Embarcado.

FR6. Identificar que o Dispositivo Embarcado consegue se comunicar com o Servidor Web.

- Componentes Relacionados: Dispositivo Embarcado e Servidor Web.
- Prioridade: Alta;
- Envolvidos: Usuário Instalador e Usuário Mantenedor;
- Descrição: Na primeira vez que o Dispositivo Embarcado é ligado, ele deve tentar se comunicar com o Servidor Web para realizar uma autenticação. Caso o dispositivo consiga realizar essa autenticação ele é considerado funcional e a instalação sucedida; caso contrário o dispositivo deve ser substituído e encaminhado para a Empresa Desenvolvedora.
- Detalhes: O Indicador Sonoro deve ser ativado por um segundo em caso de instalação sucedida; caso contrário deve ser acionado duas vezes em um segundo.

3.1.4.4 REQUISITOS DO USUÁRIO MANTENEDOR

Os requisitos FR4, FR5, FR6 são comuns ao Usuário Mantenedor.

FR7. Configurar Dispositivo Embarcado por interface serial.

- Componentes Relacionados: Dispositivo Embarcado;
- Prioridade: Alta;
- Envolvidos: Usuário Mantenedor.
- Descrição: O usuário envolvido tem interesse em realizar, ou modificar, configurações do Dispositivo Embarcado quando este sai de fábrica ou vai para manutenção.
- Detalhes: Deve ser possível modificar configurações de associação de um Dispositivo Embarcado a uma placa de veículo ao qual ele será instalado. Deve ser possível também modificar informações de conectividade na rede GPRS.

3.1.5 DIAGRAMA DE CASOS DE USO

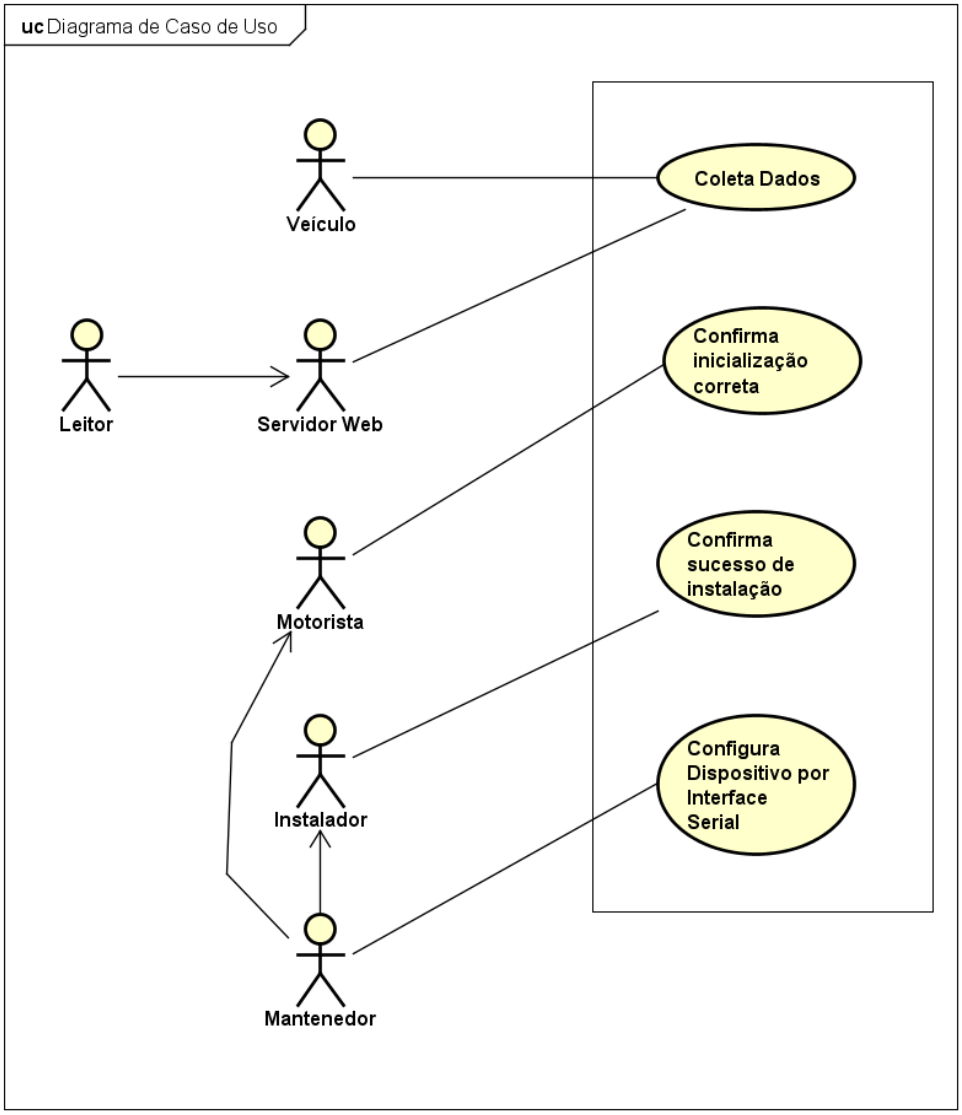
Baseado nos requisitos do projeto, foi possível definir casos de uso do Dispositivo Embarcado, que é apresentado na Fig. 11. Os passos de cada caso não são descritos nesse trabalho porque eles são relevantes para a Camada de Aplicação na arquitetura de *software*, enquanto este trabalho se limita ao desenvolvimento de

módulos na camada de Serviço e Driver (ver Arquitetura de Três Camadas no capítulo de Revisão).

É importante perceber na Fig. 11 que o usuário Leitor é visto como um usuário Servidor Web para o dispositivo embarcado. Essa unificação é feita porque o canal pelo qual aqueles usuários se comunicam com o dispositivo é através da interface web do servidor, que se comunica com o dispositivo pela rede de telefone GPRS.

O Usuário Mantenedor possui casos de uso comuns ao Motorista e ao Instalador, e portanto foi colocada uma seta indicando composição desses usuários.

FIGURA 11– DIAGRAMA DE CASO DE USO DO DISPOSITIVO EMBARCADO.



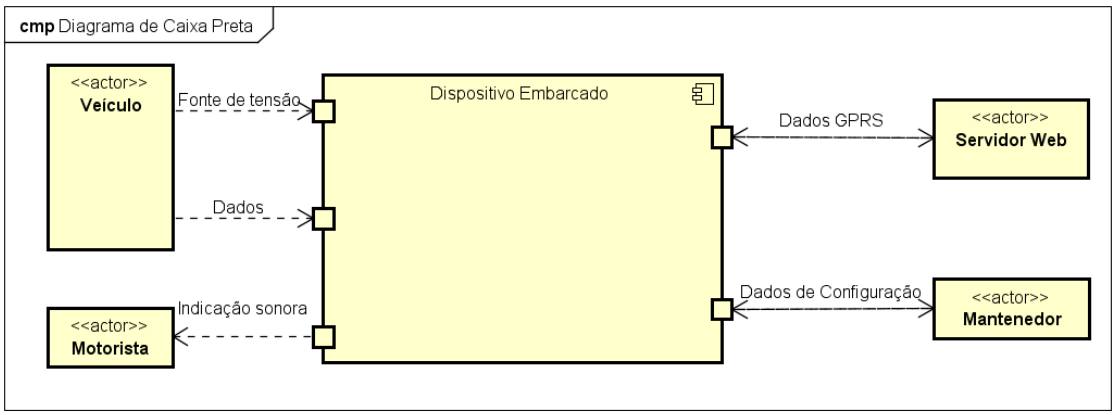
3.2 PROJETO DO SISTEMA

Na fase de *Especificação de Requisitos do Sistema* foi definido um "Modelo Independe de Computações" (MIC) para modelar o ambiente e os requisitos. A fase *System Design* refina o modelo MIC para realizar um "Modelo Independente da Plataforma" – isto é feito adicionando detalhes computacionais através da definição de componentes e uma arquitetura de alto nível. Assim, nesta fase foi desenvolvida uma arquitetura em alto nível de *hardware* e *firmware* do sistema.

3.2.1 DIAGRAMA DE CAIXA PRETA

O diagrama de caixa preta é similar a um Diagrama de Definição de Blocos, com a diferença que nesse diagrama cada bloco é um ator que interage com o Dispositivo Embarcado. O objetivo do diagrama é de explicitar a relação entre agentes externos ao Dispositivo Embarcado e os tipos de dados que são trocados. O diagrama de caixa preta do Dispositivo Embarcado é apresentado na Fig. 12. Uma vez que o diagrama é realizado, pode-se ter uma ideia melhor de quais são as interfaces entre o Dispositivo Embarcado e os atores.

FIGURA 12 – DIAGRAMA DE CAIXA PRETA DO DISPOSITIVO EMBARCADO.



FONTE: DIAGRAMA FEITO NO PROGRAMA ASTAH.

3.2.2 DIAGRAMA DE REQUISITOS

A partir dos requisitos levantados, foi possível fazer um Diagrama de Requisitos, de onde foram derivados os módulos de *hardware* necessários para o Dispositivo Embarcado. O Diagrama de Requisitos pode ser visto na Fig. 14.

3.2.2.1 CONVERSOR CC-CC

O conversor CC-CC é um módulo de *hardware* responsável por converter uma tensão de entrada DC para diversos níveis de tensões, inferiores ou superiores, com intuito de fornecer uma fonte de alimentação para todos os dispositivos do sistema embarcado.

3.2.2.2 ACELERÔMETRO

Dispositivo medidor de aceleração (força-g) no eixo vertical. Este dispositivo irá medir a intensidade de trepidações e impactos que o veículo possa vir a sofrer.

3.2.2.3 CAN

Dispositivo leitor do barramento CAN (Controller Area Network) do veículo. Este dispositivo irá ler informações de velocidade instantânea e quilometragem do veículo.

3.2.2.4 BUZZER

Dispositivo emissor de sinal sonoro de sinalização. Os sinais emitidos por esse dispositivo irão indicar funcionamento fora do normal para agentes externos.

3.2.2.5 INTERACE SERIAL

Interface de comunicação serial entre o dispositivo controlador interno ao Dispositivo Embarcado. Esta interface serial será utilizada como forma de comunicação entre o Dispositivo Embarco e agentes externos que desejem configurar o sistema.

3.2.2.6 MODEM

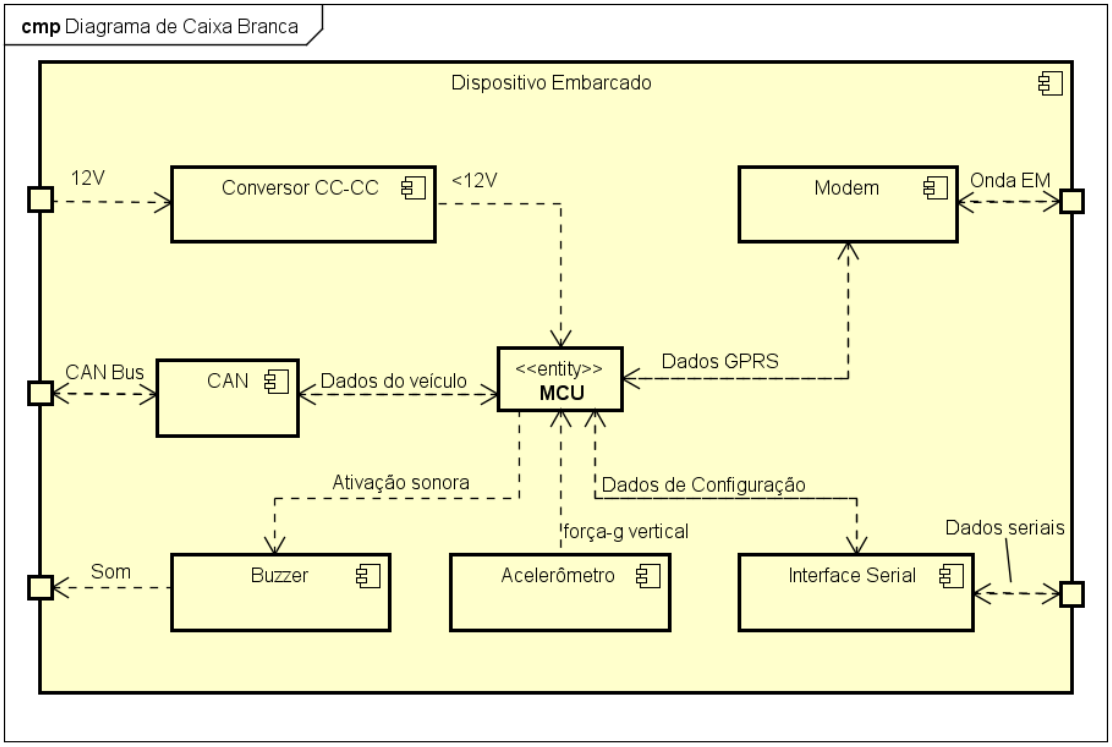
Dispositivo que realiza comunicação GPRS com um servidor externo. Este dispositivo é dotado de uma antena de transmissão GPRS.

3.2.3 PROJETO DE *HARDWARE* DO SISTEMA

Um projeto de arquitetura de alto nível foi definido com base nos requisitos do dispositivo embarcado, no diagrama de requisitos e os módulos derivados.

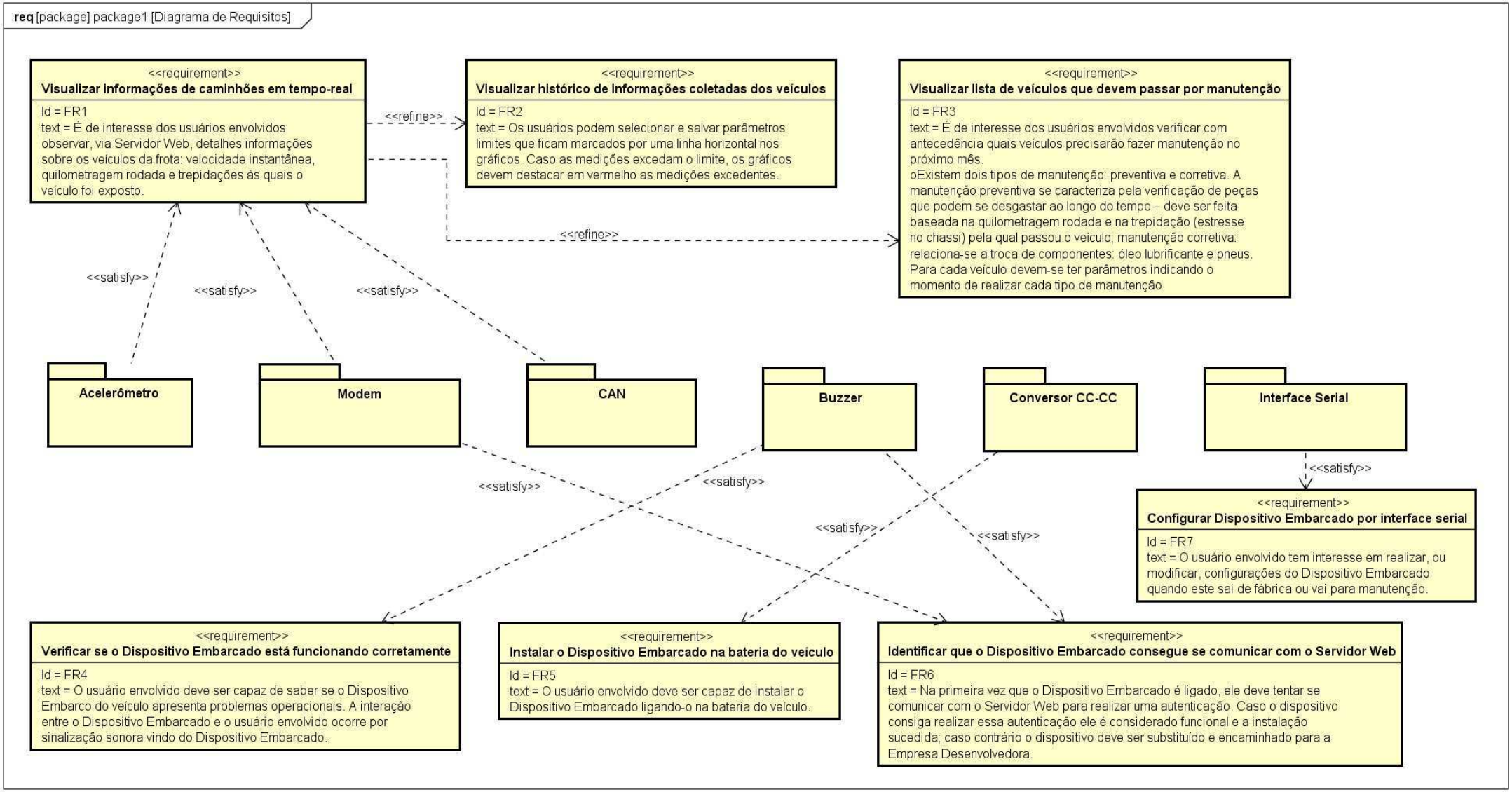
O contexto do Dispositivo Embarcado foi definido por meio de um Diagrama de Caixa Branca, e é apresentado na Fig. 13. O diagrama mostra o fluxo de dados internos ao Dispositivo Embarcado. Este diagrama inclui os módulos previstos no tópico Diagrama de Requisitos.

FIGURA 13 – DIAGRAMA DE CAIXA BRANCA DO DISPOSITIVO EMBARCADO.



FONTE: DIAGRAMA FEITO NO PROGRAMA ASTAH.

FIGURA 14 - DIAGRAMA DE REQUISITOS.



3.2.4 PROJETO DE *FIRMWARE*

A Arquitetura de *Software* escolhida para implementar o *firmware* do sistema foi a Arquitetura de Três Camadas. Os módulos de *software* que compõem a arquitetura foram escolhidos de acordo com os requisitos elucidados para o sistema. Cada módulo pode representar uma ou mais tarefas rodando concorrentemente e/ou um conjunto de APIs.

Os módulos da camada de aplicação podem ser melhores refinados com o desenvolvimento de casos de uso. Estes módulos se utilizam dos Serviços para realizarem a funcionalidade observada pelos usuários finais.

Apesar do objetivo desse trabalho ser de definir um driver para o Modem, é fundamental mostrar que outros componentes de *software* estão planejados para executar concorrentemente. A partir dessa observação, conclui-se que um Sistema Operacional é necessário para o sistema.

Um diagrama de pacotes, mostrando a interação de todos os módulos de *software* previstos para o sistema, é apresentado na Fig. 15.

3.2.4.1 BOOTLOADER

O módulo Bootloader é a porção de instruções que executa primeiro quando o dispositivo é ligado. Este módulo é responsável por inicializar Drivers considerados essenciais, e colocar a aplicação principal para executar. Este segmento de instrução deve ser programado em um local de memória diferente do resto do *firmware*, para posteriormente chamar a execução do bloco de instruções do *firmware*.

3.2.4.2 RTOS

O módulo RTOS é um Sistema Operacional em Tempo Real. Este SO deve prover funcionalidades de escalonamento de tarefas baseado em prioridades, implementações de filas e semáforos.

3.2.4.3 GPIO

O módulo Driver GPIO é responsável por operar as Portas de Entrada e Saída de Propósito Geral do Microcontrolador onde o *firmware* irá rodar. Este módulo fornece uma interface para utilização das portas por outros módulos do *firmware*.

3.2.4.4 RTC

O módulo Driver RTC é responsável por operar o Relógio de Tempo Real do Microcontrolador. Este módulo fornece uma interface para que o Serviço Timer busque valores de RTC atualizados. Caso a aplicação venha a precisar realizar interrupções de RTC, este módulo deverá prover a interface para esta funcionalidade. O Sistema Operacional utiliza o RTC através de bibliotecas do microcontrolador – as mesmas que o RTC irão utilizar.

3.2.4.5 CAN

O módulo CAN é responsável por operar o *hardware* leitor de CAN (que irá realizar a leitura do barramento CAN do veículo). Este módulo deve oferecer uma interface para leitura de mensagens com dados de um ou mais canais de comunicação CAN por outros módulos do sistema. O escopo do bloco não inclui análise do conteúdo das mensagens.

3.2.4.6 UART

O módulo UART é responsável por operar canais de UART no Microcontrolador. Este módulo realiza inicialização de canais de UART e fornece uma interface para que outros módulos façam leitura de um canal específico da UART. O escopo do bloco não inclui análise do conteúdo das mensagens.

3.2.4.7 ACELEROMETRO

O módulo Acelerometro opera o dispositivo de *hardware* Acelerômetro. Este módulo realiza acionamento e desligamento, procedimento de inicialização do *hardware* e oferece uma interface para leitura dos dados do acelerômetro por outros módulos do *firmware* em uma periodicidade estabelecida.

3.2.4.8 BUZZER

O Driver Buzzer fornece uma interface para acionamento do módulo de *hardware* Buzzer, responsável por emitir sinais sonoros.

3.2.4.9 DRIVER MODEM

O Driver Modem é um módulo de *software* responsável por operar o dispositivo Modem GPRS. Esse módulo provê uma interface independente de *hardware* (*Hardware Abstraction Layer*) para camadas acima (ou outros módulos). Esta interface fornece funcionalidades para inicialização, configuração, manipulação do modem e transmissão de dados para um servidor.

3.2.4.10 SERVIÇO DADOS_ACELEROMETRO

Este módulo é responsável por operar o Driver Acelerometro de forma a definir uma taxa de amostragem que o driver realiza, fazer leitura dos dados do driver e realizar a adequada conversão desses dados para unidades adequadas para uso dos módulos de aplicação.

3.2.4.11 SERVIÇO CAN_MAILBOX

Este módulo é responsável por operar o Driver CAN. O módulo deve ser capaz de requisitar acionamento e desligamento do leitor CAN em momentos apropriados. Deve ainda expor uma interface para abertura de canal CAN com um dispositivo externo, e realizar a uma interface de comunicação para que outros módulos possam enviar e receber mensagens através do canal CAN.

3.2.4.12 SERVIÇO TIMER

O módulo Serviço Timer é responsável por prover serviços de temporização. Estes serviços incluem, mas podem não se limitar a, buscar data e hora atuais, iniciar um temporizado (contagem regressiva), parar um temporizador, reiniciar um temporizador e buscar estado de um temporizador (iniciado, parado, expirado).

3.2.4.13 SERVIÇO BUZZER_INTERFACE

O Serviço Buzzer_Interface realiza uma interface para acionamento de Buzzer. O Buzzer_Interface oferece funcionalidade de acionamento do *hardware* buzzer de forma periódica durante um intervalo de tempo definido pelos módulos que chamam esse Serviço.

3.2.4.14 SERVIÇO SERIAL_DEBUG

O Serviço Serial_Debug oferece APIs de impressão de texto para Debug (com data e horário da impressão), e realiza a comunicação entre o Dispositivo Embarcado e um dispositivo externo para depuração do Dispositivo Embarcado, quando o último for encaminhado para manutenção.

3.2.4.15 SERVIÇO NETWORK

O Serviço Network é um módulo responsável por operar o Driver Modem. Esse módulo utiliza APIs do Driver Modem para prover serviços de comunicação via rede GPRS para outros módulos. O módulo Network é "consciente" dos procedimentos necessário para abrir um socket[5], ou registrar o módulo na rede GPRS de uma

[5] Socket: Ponto fim em uma conexão através de uma rede de computadores. Neste trabalho a rede é baseada no Protocolo de Internet.

operadora. O Serviço Network também provê APIs que informam dados operacionais do Modem, como estado de *socket*, qualidade do sinal GPRS e potência do sinal da antena do Modem.

3.2.4.16 APLICAÇÃO PRODUCER

A Aplicação Producer é responsável por colher dados dos Serviços Dados_Acelerometro e CAN_Mailbox. Esta Aplicação deve realizar uma média dos valores instantâneos medidos em um período de tempo e encapsular mensagens com estes dados para serem enviadas para o Servidor Web. Esta aplicação é "consciente" da frequência e tipos de mensagens que a Empresa Cliente espera receber. As especificações desse módulo devem ser alinhadas com os Casos de Uso (não definido neste trabalho). Uma vez que as mensagens são encapsuladas, estas devem ser enviadas, através de uma fila, para a Aplicação Producer.

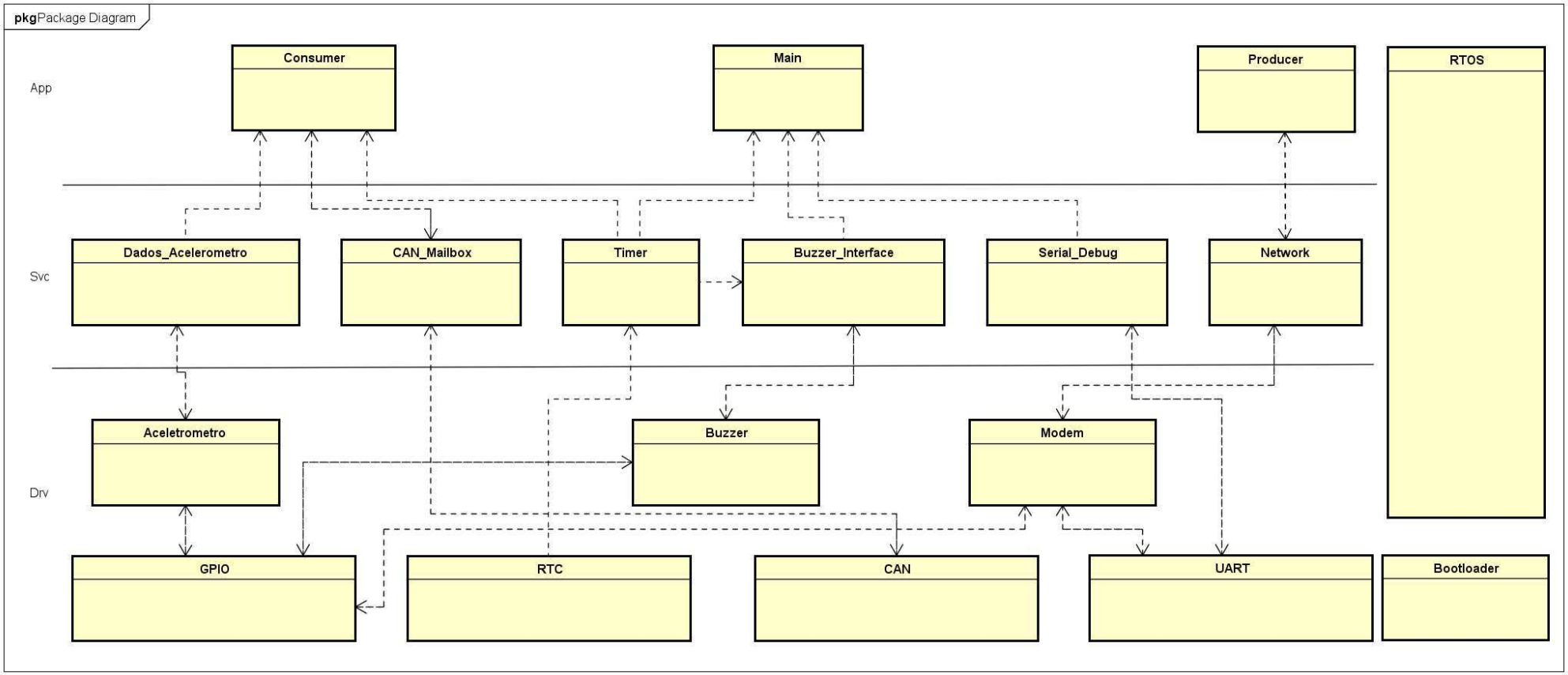
3.2.4.17 APLICAÇÃO CONSUMER

A aplicação Consumer é responsável por receber mensagens da Aplicação Producer, através de uma fila, e enviar as mensagens para o Servidor Web. O envio das mensagens é realizando por meio de APIs do Serviço Network.

3.2.4.18 APLICAÇÃO MAIN

A Aplicação Main é responsável por inicializar o ambiente do *firmware*. Esta aplicação irá acionar tarefas dos Driver, Serviços e Aplicações para começarem a rodar. Ela também realiza um auto-diagnóstico, verificando se todos os componentes estão íntegros. Também é responsável por enviar uma mensagem para o Servidor Web indicando uma primeira ativação do Dispositivo Embarcado. A Aplicação Main deve ser chamada pelo Bootloader.

FIGURA 15 - DIAGRAMA DE PACOTES DO *FIRMWARE* EMBARCADO.



3.3 PROJETO DOS COMPONENTES

A fase *Component Design* realiza a especificação interna de cada módulo demonstrado na *System Design*. Na parte de desenvolvimento de *hardware*, a fase *Component Design* é focada em escolher um modelo específico de cada componente que realizará as atribuições já descritas. Isto incluiu pesquisa de mercado envolvendo diferentes fabricantes, propriedades operacionais e preços.

Uma vez que um modelo de *hardware* tenha sido escolhido na fase *Component Design* também estuda-se as funcionalidades de cada pino do *hardware* em questão, a necessidade ou não de utilização, e como eles devem se interconectar no projeto da PCB[6] a se realizar. Os detalhes dessa fase relativa ao *hardware* não serão expostos nesse trabalho porque fogem do escopo da definição de um driver para o modem.

A operação de um modelo específico de modem GPRS não será discutida neste trabalho. Irá ser considerada a escolha de um modem que se comunica através de comunicação serial UART, via comandos atômicos enviados de um Microcontrolador para o modem em questão.

Em relação à parte de desenvolvimento de *software*, foi definida a operação dos módulos de *firmware* Driver Modem e Serviço Network. Esta fase focou em definir a necessidade e o comportamento de tarefas para esses módulos, assim como as APIs a serem implementadas. Os projetos de outros módulos não são estudados neste trabalho porque fogem do escopo planejado.

3.3.1 ESPECIFICAÇÃO DO DRIVER MODEM

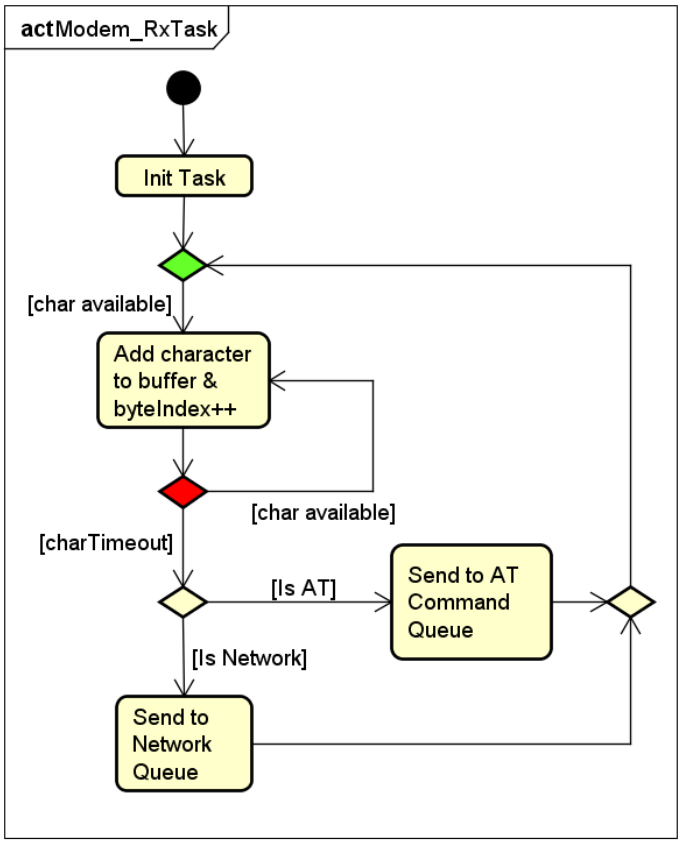
O Driver Modem implementa uma tarefa e várias APIs – além de outros métodos privados ao contexto do Modem. Os próximos subtópicos irão explicar o projeto da tarefa, um método privado utilizado de forma genérica para envio e recebimento de comandos atômicos e as APIs do driver.

[6] PCB: Printed Circuit Board (em português, Placa de Circuito Impresso)

3.3.1.1 TAREFA RX TASK

A tarefa RxTask é responsável por receber e encaminhar todos os dados que são recebidos pela porta UART do modem. A implementação desta tarefa é demonstrada na Fig. 16.

FIGURA 16 - ATIVIDADE DA TAREFA RX TASK.

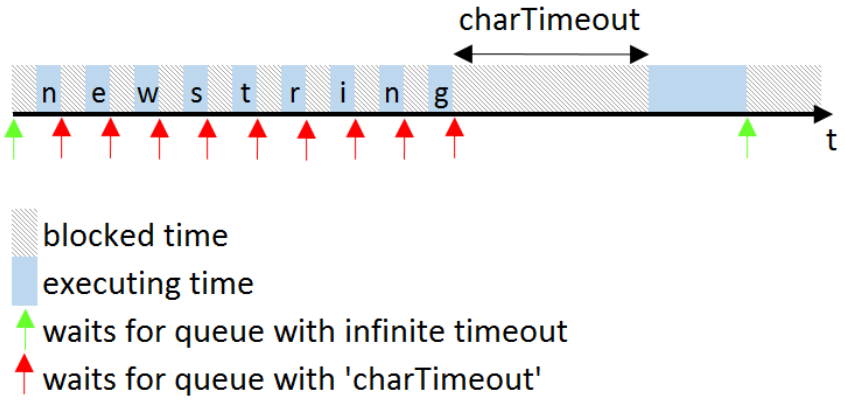


O recebimento de dados vindos da UART ocorre byte a byte, através de uma função bloqueante fornecida pela UART – esta função implementa uma fila que recebe elementos do tamanho de um byte (um caractere ASCII). Uma vez que um byte chega no canal da UART do modem, o Driver UART encaminha este byte para uma fila, e uma API do mesmo driver faz a interface para recebimento do byte através dessa fila.

O primeiro nó de decisão (verde) na Fig. 16 é um *loop while* que fica bloqueado indefinidamente esperando um byte chegar. Uma vez que um byte chega ele é adicionado a um buffer, e a execução entra em um segundo *loop* (nó de decisão vermelho). Este segundo *loop* espera por um outro byte, mas o bloqueio à espera do byte agora tem um *timeout* chamado de *characterTimeout*. Este *timeout* é longo o suficiente para a transmissão de dois bytes. Se nenhum byte for recebido nesse intervalo

de tempo, a execução sai do *loop*. Um plano de execução da tarefa Rx Task é apresentado na Fig. 17 demonstrando o recebimento de uma *string*.

FIGURA 17 – PLANO DE EXECUÇÃO DA TAREFA RXTASK RECEBENDO UMA STRING.



Após sair do segundo *loop,* o buffer é analisado: se tiver característica de comando AT, vai para uma fila de comandos AT; caso contrário vai para uma fila de dados de rede GPRS.

3.3.2.1.1 COMANDOS ATÔMICOS E APIS

Os comandos que são enviados para o modem, através do canal de UART, são chamados de comandos AT. Uma vez que o modem recebe um comando AT, ele executa o comando e responde uma mensagem indicando o resultado da execução.

Se um comando AT for enviando antes da resposta de um enviado anteriormente, os modens normalmente param de responder a qualquer comando futuro. Portanto, vê-se apropriado estabelecer um tempo de *timeout* para recebimento de comandos AT. Caso o comando não chegue no tempo máximo estabelecido, o modem deve ser reinicializado.

Dessa forma, cada comando AT pode ser encapsulado por um *struct* contendo:

- Const char * const – string com o comando AT a ser enviado;
- Uint – *timeout* em milissegundos.

Um *array* com *structs* contendo todos os comandos AT previstos na execução do programa deve ser definido. E um *enum* deve identificar o índice de cada comando.

Métodos privados que executam o envio de comandos AT devem ter a seguinte forma:

```
errorType sendATCommandProtected(uint commandId, command_t
* response, …);
```

sendATCommandProtected é o nome da função que implementa o envio de um comando AT. Os comandos AT podem receber um número variado de argumentos. Da

mesma forma, a função *sendATCommandProtected* também recebe um número variado de argumentos, que são adicionados à *string* do Comando AT que será enviado. A variável *command*_t é um *struct* que encapsula um comando AT com uma *string* e um tamanho, e o *commandId* é um ID que identifica o comando AT a ser enviado.

O método implementa tanto o envio de um comando AT quanto o recebimento da resposta do modem. Este método pega um semáforo antes de enviar o comando AT pelo canal da UART, e só o libera quando a fila de comandos AT retorna uma resposta ou o *timeout* de buscar um elemento expira. Assim, o retorno *errorType* é uma variável *enum* definida para listar diferentes erros. Caso o modem retorne algo dentro do tempo estimado de *timeout*, o *errorType* indica sucesso; caso contrário indica erro de *timeout*.

O método *sendATCommandProtected* é definido para ser utilizado de forma genérica por outros métodos que implementam envio de comando AT – sendo cada método uma abstração de uma chamada a um comando AT. Dessa forma, cada método para um comando AT irá utilizar o *sendATCommandProtected*, e analisar a resposta recebida do modem. A função terá o seguinte formato genérico:

```
errorType sendATCommand(uint argument1, argumentN, uint * result);
```

Os argumentos *argument1*, *argumentN* são previamente estabelecidos, uma vez que o desenvolvedor sabe a quantidade de argumentos que o comando AT recebe. Como alguns comandos AT retornam um resultado (ex. potência do sinal da antena), outro argumento possível é o *result*, passado por referência. Caso o modem retorne uma resposta indicando um erro na execução do comando AT, este erro deve ser indicado através do retorno *errorType*.

Considerando uma implementação no padrão C99, as seguintes APIs são esperadas para a implementação do Driver Modem (onde *error_t* é uma variável do tipo *errorType*, discutida acima):

- error_t Modem_Init (void): Essa API inicializa variáveis do contexto do Driver Modem, filas, semáforos e a tarefa __Modem_RxTask. O modem só pode iniciar sua operação uma vez que *Modem_Init()* for chamada.
- error_t Modem_Activate (void): Essa API executa o procedimento de ligar o modem. O procedimento também verifica se o modem está responsivo por meio do envio do comando AT "AT\r", esperando um OK como resposta.

- error_t Modem_Config (void): Essa API configura a operação do modem por meio da execução do envio de comandos AT de configuração (enviados por outras funções privadas).
- error_t Modem_Reset (void): Esta API realiza o procedimento de reinicialização do modem.
- error_t Modem_PowerOff (void): Essa API realiza o procedimento de desligamento do modem. Caso o modem não esteja respondendo ao comando de fazer o processador dormir, a API desliga o modem de forma forçada.
- error_t Modem_ConfigSimCardInfo (void): Essa API armazena as informações do SIM card (ICCID e IMSI) nas variáveis de contexto do Driver Modem.
- error_t Modem_RequestGprsSignal (uint8_t *result): Essa API retorna o estatus do registro do modem na rede GPRS.
- error_t Modem_GprsRegistration (uint8_t state): Essa API requisita ao modem registrar ou desregistrar o dispositivo na rede GPRS.
- uint8_t Modem_GetOperationalStatus(void): Esta API retorna o status operacional do SIM *card*.
- uint64_t Modem_GetImsi (void): Essa API retorna o número IMSI do SIM card contido no *slot* selecionado.
- error_t Modem_GetIccid(char *const iccid, uint8_t iccidSize): Essa API passa por referência o número ICCID do SIM *card*.
- uint64_t Modem_GetImei(void): Essa API retorna o número IMEI do modem.
- error_t Modem_ConfigPdpContext (const char *url): Essa API configura um contexto PDP com a URL recebida como argumento.
- error_t Modem_ActivatePdpContext(char *username, char *password): Essa API ativa o contexto PDP com o *username* e *password* enviados nos argumentos.
- error_t Modem_ConfigSocket (uint8_t connectionId, Modem_PdpContextIdType contextId, uint16_t packetSize, uint16_t exchangeTimeout, uint16_t connectionTimeout, uint16_t sendTimeout):

Essa API configura um *socket* com informações relacionadas à conexão (*timeouts* de *socket).*

- error_t Modem_OpenSocket(uint8_t protocol, uint16_t hostPort, const char *const hostIp, uint8_t closureType, uint16_t localPort): Essa API requisita ao modem a abertura de socket com um servidor remoto.
- error_t Modem_CloseSocket (uint8_t connectionId): Essa API requisita ao modem que feche a conexão de socket.
- error_t Modem_SendReceivePacket(const void *const sendPacket, uint32_t sendSize, Modem_DataPacket *receivePacket, uint32_t timeout): Essa API envia dados para um servidor remoto a partir de um socket previamente aberto, e espera uma resposta do servidor durante o *timeout* especificado.
- error_t Modem_SendPacket(const void *const sendPacket, uint32_t sendSize): Essa API envia dados para um servidor remoto através de uma conexão de socket previamente aberta.
- error_t Modem_ReceivePacket(Modem_DataPacket *receivePacket, uint32_t timeout): Essa API tenta receber dados de um servidor remoto durante o *timeout* passado por argumento.

É importante perceber que as APIs relacionadas à conexão de socket são projetadas esperando uma conexão do dispositivo embarcado com apenas um socket por vez, porque os requisitos mostraram que uma aplicação dedicada irá se conectar a apenas um servidor remoto. As APIs listadas acima são projetadas para serem operadas por um módulo único, o Serviço Network.

3.3.2.2 SERVIÇO NETWORK

O Serviço Network implementa uma tarefa, *FsmTask*, e fornece outras APIs para Serviços acima. A forma que foi implementado o módulo prevê que um ou mais módulos utilizem as APIs do Network para enviar mensagens para um servidor remoto, ou para colher informações sobre o Modem.

3.3.2.2.1 TAREFA FSMTASK

A tarefa FsmTask é responsável por operar de forma "inteligente" o modem GPRS, utilizando as APIs fornecidas pelo Driver Modem. A tarefa implementa uma máquina de estados que gerencia o momento de realizar operações que culminam na abertura do canal de envio de dados GPRS.

Existem algumas atividades que devem ser realizadas na operação do modem: Ativação do *hardware*, configuração do modo de operação do modem (o modem tem uma rotina padrão de configuração, que deve ser chamada pelo Network), configuração de APN e socket, e abertura de socket. Além disso o Serviço Network deve ser capaz de identificar quedas de socket (canal fechou) e faltas críticas do modem (travamento, desconexão) para realizar procedimentos de tentativa de reanimação do *hardware*.

Nesta etapa do projeto surgiram outros requisitos de sistema, considerados não-funcionais:

NFR1. No começo da execução da máquina da tarefa Network, deve ser verificado se o SIM card está operacional, e as informações de ICCID e IMSI do SIM *card* devem ser requisitadas e registradas nas variáveis de contexto do Serviço Network;

NFR2. A tentativa de abrir socket só deve ser realizada uma vez que haja uma requisição de abertura do canal GPRS por outros módulo;

NFR3. O Serviço Network deve tentar se registrar na rede GPRS a cada 5 segundos;

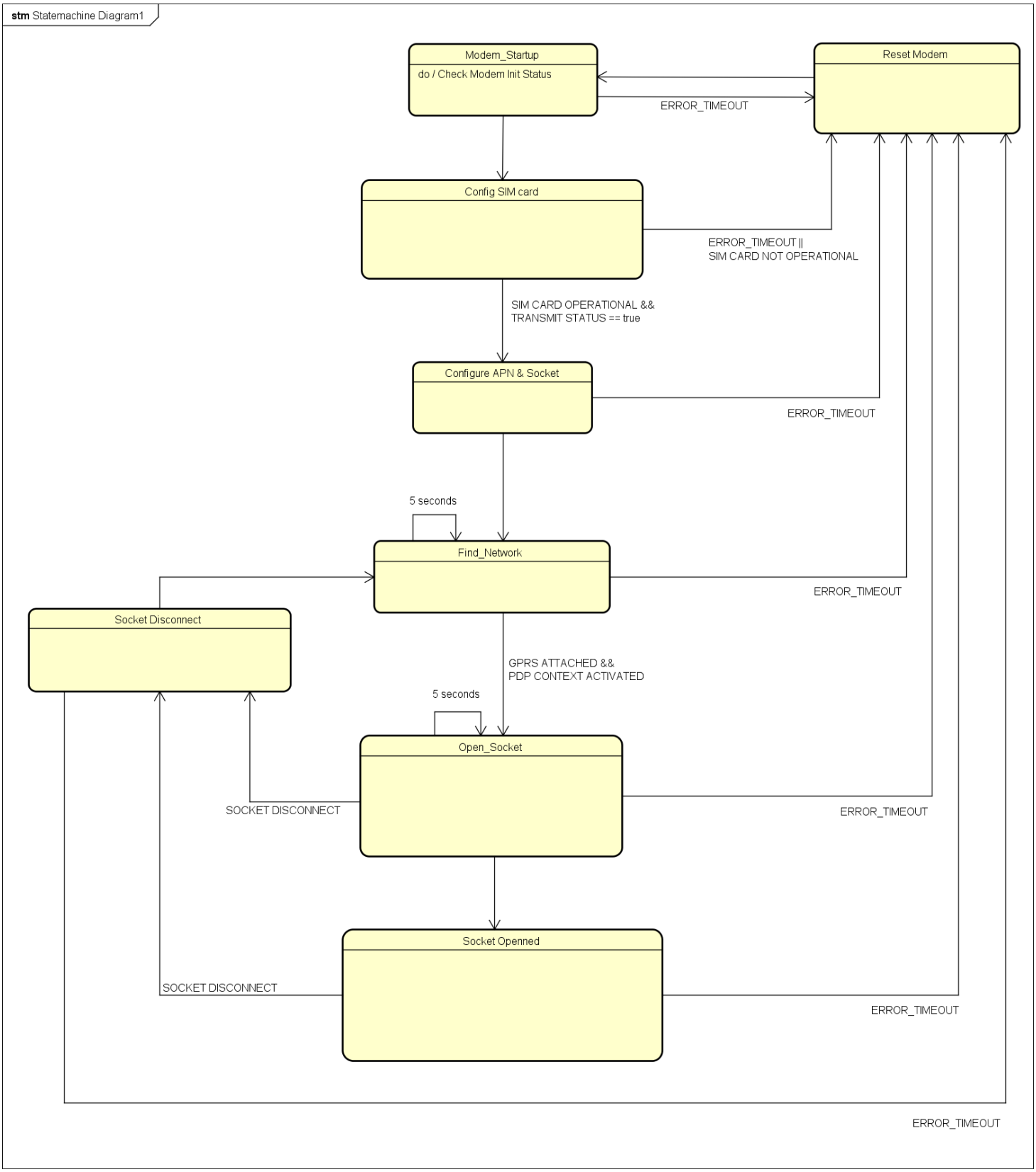
NFR4. O Serviço Network deve tentar abrir socket com o servidor remoto a cada 5 segundos.

NFR5. Durante qualquer estado do Network, se uma API chamada do Driver Modem retornar erro de *timeout*, a rotina de reinicialização do modem deve ser chamada pelo Network.

Estes requisitos são cruciais para a modelagem do comportamento da máquina de estados implementada.

O arquivo de configuração do Serviço Network registra possíveis módulos que podem requisitar abertura de socket, e requisição de entrar em estado *sleep*. Cada módulo registrado fornece um apontador para uma função que retorna verdadeiro caso haja uma requisição daquele módulo. Assim, na máquina de estado da Fig. 18, *TRANSMIT STATUS == TRUE* ocorre quando uma das funções de requisição de transmissão retornar verdadeiro. O mesmo ocorre para *SLEEP REQUEST*.

FIGURA 18 - MÁQUINA DE ESTADOS IMPLEMENTADA PELO SERVIÇO NETWORK.



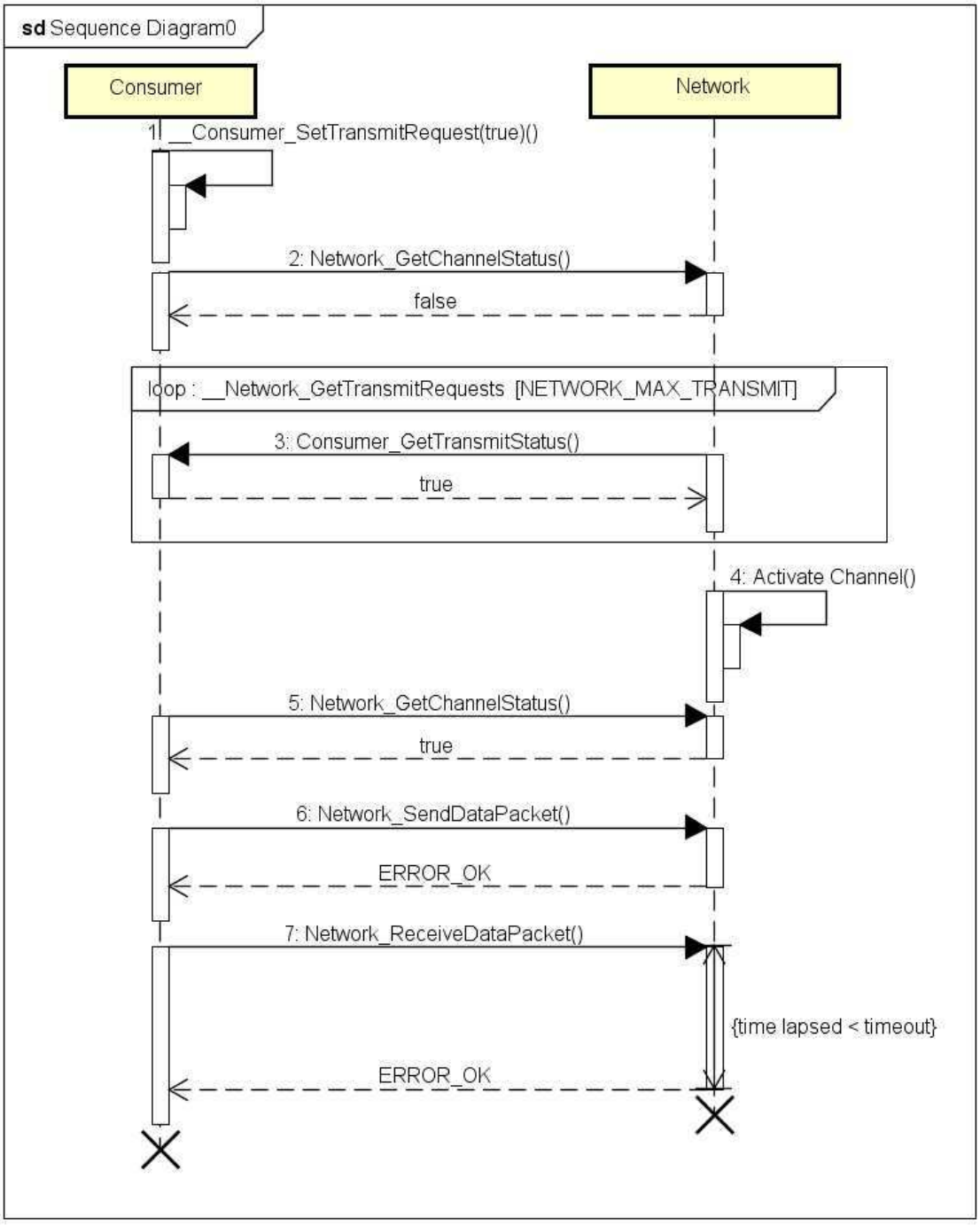
3.3.2.2.2 APIs

O Serviço Network fornece APIs de inicialização do módulo, estado do canal GPRS (aberto se a conexão de socket foi estabelecida, caso contrário fechado), e envio e recebimento de dados GPRS. Abaixo segue a descrição das APIs.

- error_t Network_Init (void): Essa API inicializa variáveis do contexto do Serviço Network, filas, semáforos e a tarefa __Network_FsmNetwork, que executa a máquina de estados do Network;
- uint8_t Network_GetChannelStatus (void): Essa API retorna o estado do canal de comunicação GPRS. Verdadeiro indica aberto, e falso indica fechado;
- error_t Network_SendReceivePacket (void *sendData, uint32_t sendSize, Modem_DataPacket *receivedData): Essa API envia uma mensagem através do canal de dados GPRS para um servidor remoto. A chamada é bloqueante pelo período de tempo NETWORK_RX_DATA_EXCHANGE_TIMEOUT_ms, definido no arquivo de configuração do Network;
- error_t Network_SendPacket (void *sendData, uint32_t sendSize): Essa API envia uma mensagem através do canal de dados GPRS para um servidor remoto.
- error_t Network_ReceivePacket (void *buffer, uint16_t length, uint16_t *copiedSize): Essa API tenta receber uma mensagem pelo canal de dados GPRS por um período de tempo (bloqueante) NETWORK_RX_DATA_EXCHANGE_TIMEOUT_ms, definido no arquivo de configuração do Network;
- uint64_t Network_GetModemImei (void): Essa API retorna o número IMEI do modem;
- error_t Network_GetSimCardIccid (uint8_t slot, char *const iccid, uint8_t iccidSize): Essa API passa por referência o número ICCID do SIM card escolhido;
- uint64_t Network_GetSimCardImsi (Modem_SimCardType slot): Essa API retorna o número IMSI do SIM card selecionado;

Um diagrama de sequência mostrando o procedimento necessário para um módulo enviar dados pelo canal GPRS é apresentado na Fig. 19.

FIGURA 19 – DIAGRAMA DE SEQUÊNCIA PARA TRANSMISSÃO DE DADOS PELO CANAL GPRS DO SERVIÇO NETWORK.



4 CONCLUSÃO

O Desenvolvimento Baseado em Modelos, aliado com a técnica de desenvolvimento de projetos baseada no diagrama V-Model, mostrou-se de suma importância para desenvolvimento e documentação do projeto realizado neste trabalho. A utilização do diagrama V-Model foi essencial para o desenvolvimento de uma solução guiado por etapas definidas.

A utilização de diagramas durante o desenvolvimento também é um diferencial, pois deixa documentado de forma clara o funcionamento do projeto. Novos membros que necessitem utilizar partes do projeto, ou fazer atualizações, podem partir para fases de implementação com maior agilidade e sem ocupar o tempo de outros funcionários de uma empresa para tirar dúvidas sobre o funcionamento do projeto.

A utilização de um Sistema Operacional em Tempo Real se torna muito útil para realizar a sincronização de tarefas concorrentes. Recursos tipicamente oferecidos pelo SOs e RTOSs, como filas e semáforos, são essenciais para sincronizar as ações de cada módulo de *software* previsto. O projeto realizado neste trabalho foi baseado em modelos independentes de tipo de Sistema Operacional que deseje-se utilizar. Na fase de levantamente de requisitos foi prevista a implementação em um sistema embarcado, isto limita as opções de sistemas operacionais. Entretanto o projeto pode ser implementado em diversas plataformas de *hardware*, sendo mais comum a utilização em dispositivos microcontroladores.

REFERÊNCIAS

de Almeida e Bueno, L., Barros Zanoni Lopes Moreno, G., & Sweet, B. (2013). Three-Layer Software Architecture Inspired by AUTOSAR applied in a Telemetry System for a Radio-Controlled Aircraft, 2013 IEEE INTERNATIONAL CONFERENCE on ELECTRO/INFORMATION TECHNOLOGY. Rapid City: [s.n.]. 2013.

Friedenthal, S., Moore, A., & Steiner, R. (acessado em 27/11/2015). *About Us: The Object Management Group (OMG).* Retrieved from The Official OMG SysML site: http://www.omgsysml.org/SysML-Tutorial-Baseline-to-INCOSE-060524-low_res.pdf

Gomma, H. (2011). *Software Modeling & Design.* New York: cambridge university press.

Isovic, D. (2012). *Introduction to Real-Time Systems.* Mälardalen University, Disciplina Embedded Systems 2, Västerås.

Li, Q., & Yao, C. (2003). *Real-Time Concepts for Embedded Systems.* San Francisco, CA: CMP Books.

Scippacercola, F., Pietrantuono, R., Russo, S., & Zentai, A. (09/10/2015). Model-Driven Engineering of a Railway Interlocking System. *ResearchGate*, p. 12.

Stallings, W. (2008). *Operating Systems: Internals and Design Principles (*5ª ed*.).* Pearson.

Vermesan, O., & Friess, P. (30/12/2014). *Internet of Things – From Research and Innovation to Market Deployment* (2ª ed.). Aalborg, Brasil: River Publishers.