Rodrigo Valceli Raimundo

Interoperabilidade de sistemas embarcados com uma pilha de protocolos flexível, configurável e de baixo custo

Florianópolis

Outubro de 2011

Rodrigo Valceli Raimundo

Interoperabilidade de sistemas embarcados com uma pilha de protocolos flexível, configurável e de baixo custo

Orientador:
Prof. Dr. Antônio Augusto M. Fröhlich

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Florianópolis

Outubro de 2011

Monografia sob o título “Interoperabilidade de sistemas embarcados com uma pilha de protocolos flexível, configurável e de baixo custo”, defendida em XX de XXX de 2011 por Rodrigo Valceli Raimundo, como parte dos requisitos à obtenção do grau de Bacharel em Ciência da Computação, e aprovada pela seguinte banca examinadora:

Prof. Dr. Antônio Augusto M. Fröhlich
Orientador

Prof. Dr. Frank Siqueira

Prof. Dr. Mario Dantas

Resumo

A tendencia atual da Internet é permear todos os objetos e interações da vida cotidiana. Embora a conectividade com a grande rede seja assunto resolvido para uma vasta gama de sistemas computacionais, temos nichos crescentes de objetos do dia a dia que estão adquirindo maiores funcionalidades em virtude do avanço constante da eletrônica embarcada. Neste trabalho apresentamos nossa solução de conectividade IP desenvolvida na plataforma EPOS, demonstrando a viabilidade de interconexão usando protocolos maduros e bem estabelecidos mesmo em sistemas altamente reduzidos e especializados.

Abstract

The Internet's current trend is to permeate all objects and interactions of everyday life. Although the connectivity to the Internet is a finished subject for a wide range of computer systems, we have growing niches of everyday objects that are getting more functionality due to the constant advancement of embedded electronics. We present our solution for IP connectivity in our EPOS development platform, demonstrating the feasibility of using mature and well established interconnection protocols even in highly reduced and specialized systems.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de abreviaturas e siglas

1 Introdução p. 11

2 Estado da Arte p. 15

2.1 TCP/IP para sistemas embarcados . . . . . p. 15

2.2 TCP/IP em sistemas convencionais . . . . . p. 18

3 Arquitetura p. 20

3.1 Modelo . . . . . p. 20

3.2 Comportamento . . . . . p. 24

3.3 Configurabilidade . . . . . p. 32

4 Resultados p. 33

5 Conclusão p. 36

5.1 Trabalhos futuros . . . . . p. 36

Referências Bibliográficas p. 38

Apêndice A – Infra-estrutura p. 41

A.1 Cabeçalhos . . . . . p. 41

A.2 Código-fonte . . . . . p. 63

A.3 Classes utilitárias . . . . . p. 90

A.4 Testes . . . . . p. 94

Lista de Figuras

3.1 Diagrama de classes simples do modelo orientado a eventos . . . . . p. 22

3.2 Diagrama das classes relacionadas ao padrão Publish/Subscribe . . p. 23

3.3 Diagrama de classes simples do modelo síncrono . . . . . p. 24

3.4 Diagrama de atividade do recebimento de pacotes pela camada de rede. . . . . p. 26

3.5 Diagrama de atividade do envio de pacotes pela camada de rede. . . p. 27

3.6 Esqueleto da classe UDP. . . . . p. 29

3.7 Interface de programação UDP::Socket e UDP::Channel. . . . . p. 29

3.8 Recebimento de dados no TCP com janela zero . . . . . p. 31

Lista de Tabelas

4.1 Tamanho total dos arquivos-objeto de cada componente para arquitetura Intel x86 utilizando Ethernet e ARP. . . . . p. 33

4.2 Tamanho total dos arquivos-objeto de cada componente para arquitetura ARM utilizando ADHOP e CMAC. . . . . p. 34

4.3 Tamanho total dos arquivos-objeto de cada aplicativo de teste. . . . . p. 34

4.4 Memória ocupada por cada instancia das classes da hierarquia de protocolos. . . . . p. 34

Lista de abreviaturas e siglas

LISHA	Laboratório de Integração Software/Hard-ware,	p. 12
TCP	Transmission Control Protocol,	p. 15
IP	Internet Protocol,	p. 15
VoIP	Voice over Internet Protocol,	p. 15
IP-TV	Internet Protocol Television,	p. 15
ROM	Read-Only Memory,	p. 16
UDP	User Datagram Protocol,	p. 16
API	Application Programming Interface,	p. 16
BSD	Berkeley System Distribution,	p. 16
DNS	Domain Name System,	p. 18
LDAP	Lightweight Directory Access Protocol,	p. 18
EPOS	Embedded Parallel Operating System,	p. 20
ADESD	Application-Driven Embedded Software Design,	p. 20
ICMP	Internet Control Message Protocol,	p. 21
OSI	Open Systems Interconnection,	p. 25
ISR	Interrupt Service Routine,	p. 25
MSS	Maximum Segment Size,	p. 31
ARP	Address Resolution Protocol,	p. 33
C-MAC	Configurable Medium Access Protocol,	p. 33
ADHOP	Ant-based Dynamic Hop Optmization Protocol,	p. 33
GCC	GNU Compiler Collection,	p. 33
DHCP	Dynamic Host Configuration Protocol,	p. 33
TCC	Trabalho de Conclusão de Curso,	p. 36
6lowPAN	IPv6 for low power local area networks,	p. 36

1 Introdução

Comunicação de dados é um dos pilares da Ciência da Computação e a pesquisa nesta área, como disciplina científica, nos remete no mínimo à invenção do telégrafo. Porém não é por ser um campo de pesquisa antigo que seus assuntos estejam esgotados. Há pelo menos trinta anos vivemos uma explosão na dimensão e alcance dos sistemas de telecomunicação, com um forte destaque para a Internet. O que no final da década de 70 se limitava aos poucos (e enormes) computadores das principais universidades dos Estados Unidos, hoje está presente em uma parcela significativa dos lares, empresas e demais organizações. Aquilo que começou com o intuito de sobreviver a grandes calamidades hoje é tão onipresente e complexo que até mesmo os especialistas tem dificuldade em definir suas dimensões.

Em paralelo ao crescimento explosivo das redes de computadores também houve decréscimo exponencial no custo e tamanho do computadores. Chegamos na fronteira tecnológica da revolução dos circuítos integrados de larga escala e alta densidade. Tal redução levou ao surgimento dos denominados sistemas embarcados, algumas vezes chamados de sistemas embutidos ou dedicados. Muitos são os objetivos dos sistemas embarcados atuais e talvez a maior semelhança entre todos eles é de que as pessoas não os veem como um computador no seu sentido convencional, como as estações de trabalho fixas e portáteis. O encolhimento dos dispositivos computacionais e seu barateamento também possibilitou a inserção de um computador onde antes não se imaginava que fosse possível e/ou necessário. Quem alguns anos atrás acharia viável colocar um computador em cada tomada, lâmpada, ar condicionado, vestuário e até

mesmo dentro de nosso corpo? Hoje a ideia não soa tão absurda assim.

A união dessas duas revoluções alimenta nossa pesquisa atual de diversas maneiras. Atualmente temos pesquisa de ponta em MANETs (Redes de agentes móveis) e WSNs (Redes de sensores sem-fio), por exemplo. Porém o rumo atual nos leva a uma ideia muito mais abrangente. Interconectar todos os objetos do mundo criaria o que chamamos de IoT (Internet das Coisas, do inglês Internet of Things). Alguns já pensam um pouco mais além, na Web of Things (Web das Coisas). A estrutura de software e hardware desenvolvida para a Internet de super-computadores e desktops se mostrou muito custosa e inapropriada para os novos sistemas computacionais extremamente reduzidos e com diversas restrições que não se aplicavam aos sistemas computacionais tradicionais como: consumo de energia, dimensões e custo. Tais limitações fizeram surgir diversos protocolos e implementações voltados para esse novo mundo.

A união dos eventos acima, dentro do contexto do LISHA (Laboratório de Integração Software/Hardware) da UFSC e cujo projeto de longo prazo EPOS (Embedded Parallel Operating System) serve como base para nossa pesquisa em redes e sistemas embarcados, levou ao desenvolvimento do presente trabalho. O objetivo central deste trabalho é projetar e desenvolver a infra-estrutura necessária de comunicação para que dispositivos embarcados com aplicações desenvolvidas com o EPOS possam se comunicar com os demais dispositivos da Internet. Para cumprir tal objetivo é preciso aderir às normas já estabelecidas de comunicação para Internet, a famosa pilha de protocolos TCP/IP, provendo os recursos necessários para o desenvolvimento de aplicações com mínimo custo de recursos computacionais, como uso de CPU, memória e tamanho de código, além de portabilidade entre as diversas arquiteturas de hardware disponíveis no mercado.

Este trabalho está organizado da seguinte maneira. O capítulo inicial faz uma revisão do estado da arte e enumera os principais conceitos envolvidos ao longo do

texto. O capítulo seguinte mostra a arquitetura desenvolvida em detalhes. O capítulo Resultados faz uma análise comparativa entre nosso projeto e outros relacionados. Por fim temos as conclusões e ideias para trabalhos futuros.

A human being should be able to change a diaper, plan an invasion, butcher a hog, conn a ship, design a building, write a sonnet, balance accounts, build a wall, set a bone, comfort the dying, take orders, give orders, cooperate, act alone, solve equations, analyze a new problem, pitch manure, program a computer, cook a tasty meal, fight efficiently, die gallantly. Specialization is for insects. (Robert Heinlein)

2 Estado da Arte

> *The psychological profiling [of a programmer] is mostly the ability to shift levels of abstraction, from low level to high level. To see something in the small and to see something in the large.* (Donald Knuth)

A pilha de protocolos TCP/IP se consolidou como o padrão *de-facto* para comunicação de dados (DUNKELS et al., 2004), atingindo atualmente áreas já então consolidadas antes do surgimento da Internet, como a telefonia (VoIP) e a televisão (IP-TV) (GOODE, 2002). A pesquisa em intercomunicação IP no mundo dos sistemas embarcados é recente e ainda encontra muitos desafios, porém acredita-se que o futuro está na interconexão de todos os objetos do dia-a-dia (VASSEUR; DUNKELS, 2010). O objetivo desse capítulo é levantar os modelos e implementações com enfase nesta categoria de sistemas. A literatura explicando os detalhes dos protocolos TCP/IP é extensa e ficaria muito massante explicar tais conceitos básicos neste trabalho. Caso necessário uma abordagem didática pode ser encontrada em Tanenbaum (2002).

2.1 TCP/IP para sistemas embarcados

2.1.1 uIP e lwIP

Com o intuito de permitir conectividade IP a microcontroladores surgiu a primeira pilha TCP/IP para arquiteturas de 8-bit, chamada de uIP (micro-IP) e apresentada no trabalho de Dunkels (2003). Para a implementação do uIP se concretizar foram aplicadas várias limitações para minimizar o uso de memória, entra elas podemos citar:

1. Nenhuma alocação dinâmica, número máximo de conexões limitado em tempo de compilação,
2. *Buffer* de recebimento e envio único e compartilhado para ambas operações,
3. Sem demultiplexação automática,
4. Retransmissão assistida pela aplicação,
5. Sem controle de fluxo, apenas um segmento enviado por vez,
6. Suporte a uma única interface de rede,
7. Implementação monolítica desde o encapsulamento da camada de enlace até a camada de aplicação.

Tais limitações fizeram o uIP alcançar um tamanho total o pequeno suficiente para rodar em microcontroladores com 8kb de ROM. O autor afirma ainda que a portabilidade é alta, já que o código é C-puro e a interface externa necessita apenas de operações básicas de *send/receive* em uma interface de rede e um temporizador para retransmissões. Por outro lado, o esforço do lado do desenvolvedor de aplicações é maior que em outras implementações da pilha TCP/IP. Lidar com múltiplas conexões ou protocolos distintos, como UDP e TCP simultaneamente, se torna bastante complicado com a API limitada oferecida pelo uIP.

Outro esforço do mesmo autor foi a criação do lwIP (lightweight-IP), publicada no mesmo trabalho (DUNKELS, 2003) e presente no sistema operacional Contiki (DUNKELS; GRÖNVALL; VOIGT, 2004). A lwIP resolve a maior parte das limitações do uIP, com um consumo de recursos um pouco maior. A API da lwIP, embora lembre a implementação tradicional de *sockets* do Unix BSD, é feita utilizando o conceito de *protothreads*, também do mesmo autor (DUNKELS; SCHMIDT, 2005). As *protothreads* ajudam a programação orientada a eventos no Contiki, permitindo expressar de forma sequencial procedimentos que são na realidade passos de uma máquina de

estados. Tal abordagem é bem vinda no contexto do Contiki, já que o paradigma do mesmo é orientação a eventos e ele não possui suporte nativo a threads.

2.1.2 lyraNET

Algumas aplicações embarcadas, com menos restrições de memória, podem optar por executar sistemas operacionais de propósito geral, como o Linux ou o uCLinux. A lyraNet (LI; CHIANG, 2005) foi desenvolvida como uma alternativa a pilha TCP/IP do Linux, com o objetivo de reduzir ao mínimo o número de operações de cópia de memória. Tal objetivo é justificado pelo fato da maior parte do tempo de processamento das operações de rede serem gastos em operações de cópia entre a memória de kernel e a memória de usuário.

A lyraNET consegue ao mesmo tempo oferecer tanto um desempenho melhor (tempo de execução), quanto um *overhead* menor de memória em relação pilha tradicional do Linux. Embora do ponto de vista da aplicação nada seja alterado, a adapção da lyraNET para uma nova plataforma requer também a adaptação dos drivers das interfaces de rede.

2.1.3 nanoIP, nanoTCP e nanoSLP

Embora possuam IP e TCP no nome, a pilha de protocolos nanoIP (SHELBY et al., 2003) não respeita os formatos de cabeçalho definidos para os protocolos da Internet (POSTEL, 1981a; POSTEL, 1981b; POSTEL, 1980). O objetivo da pilha nanoIP foi levar alguns recursos do TCP/IP convencional para redes de sensores ou outros sistemas reduzidos, principalmente os que usam o padrão IEEE 802.15.4 para comunicação sem-fio. O nanoIP não suporta roteamento diretamente, apenas com a ajuda de *gateways* especializados. A noção de endereçamento da camada de rede foi eliminada, sendo utilizado apenas o endereçamento da camada de enlace. Todas as modificações e reduções impostas nos cabeçalhos foram para diminuir o peso do

nanoIP nos pacotes IEEE 802.15.4, que transporta no máximo 127 octetos por pacote.

Um ponto interessante da proposto do nanoIP, foi o nanoSLP. O nanoSLP é um protocolo de localização de serviços baseado em consultas textuais. Sua criação foi uma tentativa dos autores de conseguirem algo análogo ao *DNS* (MOCKAPETRIS, 1987) e aos *Directory Services*, como o LDAP (HOWES; SMITH, 1995), para as redes *mesh* construídas com nanoIP.

2.1.4 Implementações proprietárias

Além das pilhas TCP/IP citadas acima, existe um mercado de implementações proprietárias (algumas gratuitas, outras não) consolidado. A situação mais recorrente neste cenário é o fornecimento por parte dos fabricantes de microcontroladores, como a Texas Instruments, de kits de desenvolvimento de software contendo pilhas TCP/IP completas. Além do NDK (Network Development Kit) da TI, podemos citar o uC/TCP da Micrium, o MISRA TCP/IP da empresa HCC Embedded, IPLite da Interpeak, entre outros. Devido as restrições de disponibilidade destas soluções não é possível fazer uma boa análise apenas baseada em *overviews* disponibilizados pelos fabricantes. O fato de existirem diversas ofertas de soluções pagas ajuda a consolidar a importância do nicho de aplicação explorado neste trabalho.

2.2 TCP/IP em sistemas convencionais

A história de praticamente todas as implementações do protocolo TCP/IP para sistemas convencionais remota a história do BSD, o sistema Unix melhorado e re-distribuído pela Universidade da Califórnia). Um fato histórico importante é de que, o código Unix cedido pela AT&T possuía um alto custo de licenciamento, mas o código responsável pelos protocolos de rede foram desenvolvidos completamente por Berkeley e distribuídos por uma licença de software livre. Tal disponibilidade de uma implementação de qualidade e livre fez com que os demais produtores de sistemas

operacionais adotassem a implementação oriunda do BSD. O efeito prático disto foi uma homogenização da API e uma rápida difusão do TCP/IP frente aos concorrentes de sua época.

As duas primeiras implementações BSD (TCP-Tahoe e TCP-Reno) focaram em controle de congestionamento. Seus descendentes modernos como TCP CUBIC (HA; RHEE; XU, 2008) utilizado no Linux e o Coumpond TCP (TAN; SONG, 2006) utilizado a partir do Windows Vista, focaram em melhorias como *fairness* e aproveitamento máximo de banda disponível. Outras implementações como TCP SACK e TCP Fast Reno também alcançaram boa popularidade. O trabalho publicado por Mascolo (2006) mostra que tais mecanismos de controle de congestionamento podem ser descritos a partir de pequenas modificações em um mesmo modelo matemático. Uma comparação entre New Reno, CUBIC e Compound feita por Abdeljaouad et al. (2010) mostra resultados parecidos de desempenho tanto para conexões cabeadas quanto para conexões sem-fio entre as diversas implementações.

3 Arquitetura

> *The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.* (Fred Brooks, No Silver Bullet)

Este capítulo aborda as decisões tomadas na elaboração e desenvolvimento de uma pilha TCP/IP para o EPOS. Incluindo a interação com os demais componentes do sistema, a organização interna, utilitários desenvolvidos e interface de programação para os desenvolvedores de aplicativos.

3.1 Modelo

Atualmente dois modelos distintos de programação são amplamente difundidos para a implementação de APIs/frameworks de comunicação de dados. O primeiro é o modelo orientado a eventos, um modelo de computação reativo onde o usuário (programador) da infraestrutura de comunicação interage com o meio de comunicação escrevendo procedimentos conhecidos como *callbacks*. Tais callbacks são executados automaticamente pelo sistema quando um evento associado com eles for disparado. No caso de comunicação, eventos comuns são a chegada de dados, o estabelecimento e término de conexões, por exemplo. O modelo orientado a eventos está mais próximo do paradigma descritivo de programação, já que o programador diz o que deve ser executado, mas não sabe necessariamente quando será executado. De outro lado, o segundo modelo, síncrono/sequencial está relacionado com a programação procedural e de certa forma é mais difundido e utilizado. Uma API de comunicação

síncrona possui funções de envio/recebimento, estabelecimento de conexão e afins, que são usadas pelo programador de maneira sequencial dentro do fluxo da aplicação.

Neste trabalho ambos os modelos, orientado a eventos e sequencial, foram usados para construir uma interface de programação que possa ser utilizada de acordo com as necessidades da aplicação, seguindo a metodologia ADESD (Application-Driven Embedded Software Design) (FRÖHLICH, 2001).

3.1.1 Descrição do modelo orientado a eventos/assíncrono

A hierarquia de classes do modelo orientado a eventos está exibida na figura 3.1. No topo da hierarquia temos a classe NIC, que representa a interface de rede e em ADESD, é classificada como um mediador de hardware (POLPETA; FRÖHLICH, 2004). O objetivo do mediador é fornecer uma interface homogênea para utilização de dispositivos de hardware. Diferente dos tradicionais *device drivers*, o mediador é construído de forma que seu código possa ser incorporado pelos demais componentes do sistema sem a necessidade de interfaces pesadas como as *syscalls* utilizadas em boa parte dos sistemas operacionais.

Após a classe NIC, temos as classes intermediárias da infraestrutura: IP, UDP, ICMP e TCP. A divisão dos diversos elementos da camada de rede e camada de transporte em classes distintas foi feita para aumentar o desacoplamento, permitindo que os componentes possam ser usados de acordo com sua necessidade, sem apresentar adição de código não-usado no binário final de uma aplicação. Tal separação também permite um melhor entendimento, depuração e separação de conceitos dentro do código-fonte. A ligação entre as classes das diversas camadas é feita utilizando o padrão de projeto *Publish/Subscribe* (EUGSTER et al., 2003), fornecido pelas classes utilitárias Conditional_Observer, Conditional_Observer, Data_Observer e Data_Observed, apresentadas nos diagramas da figura 3.2.

Na ponta da hierarquia, temos as classes UDP::Socket e TCP::Socket, que devem

ser utilizadas pelo desenvolvedor das aplicações através de herança, implementados os métodos necessários para recepção de dados e controle de conexão no caso do TCP::Socket.

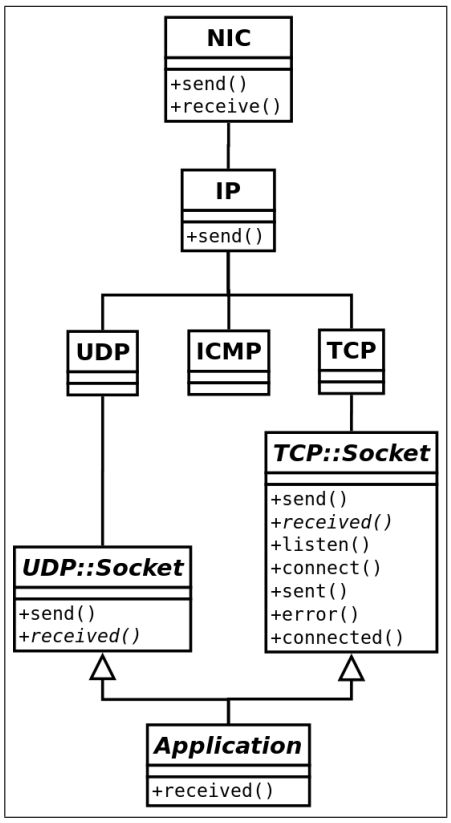

Figura 3.1: Diagrama de classes simples do modelo orientado a eventos

Vantagens e desvantegens

A API orientada a eventos deste trabalho permite a comunicação utilizando UDP e TCP com mínimo *overhead*, tanto de código, quanto de memória e tempo de CPU. Tal desempenho porém é sacrificado pelo fato de que o desenvolvimento de aplicações é significativamente mais difícil. Nossa implementação orientada a eventos segue um modelo parecido com o proposto por Dunkels (2009), onde as retransmissões devem ser assistidas pela aplicação.

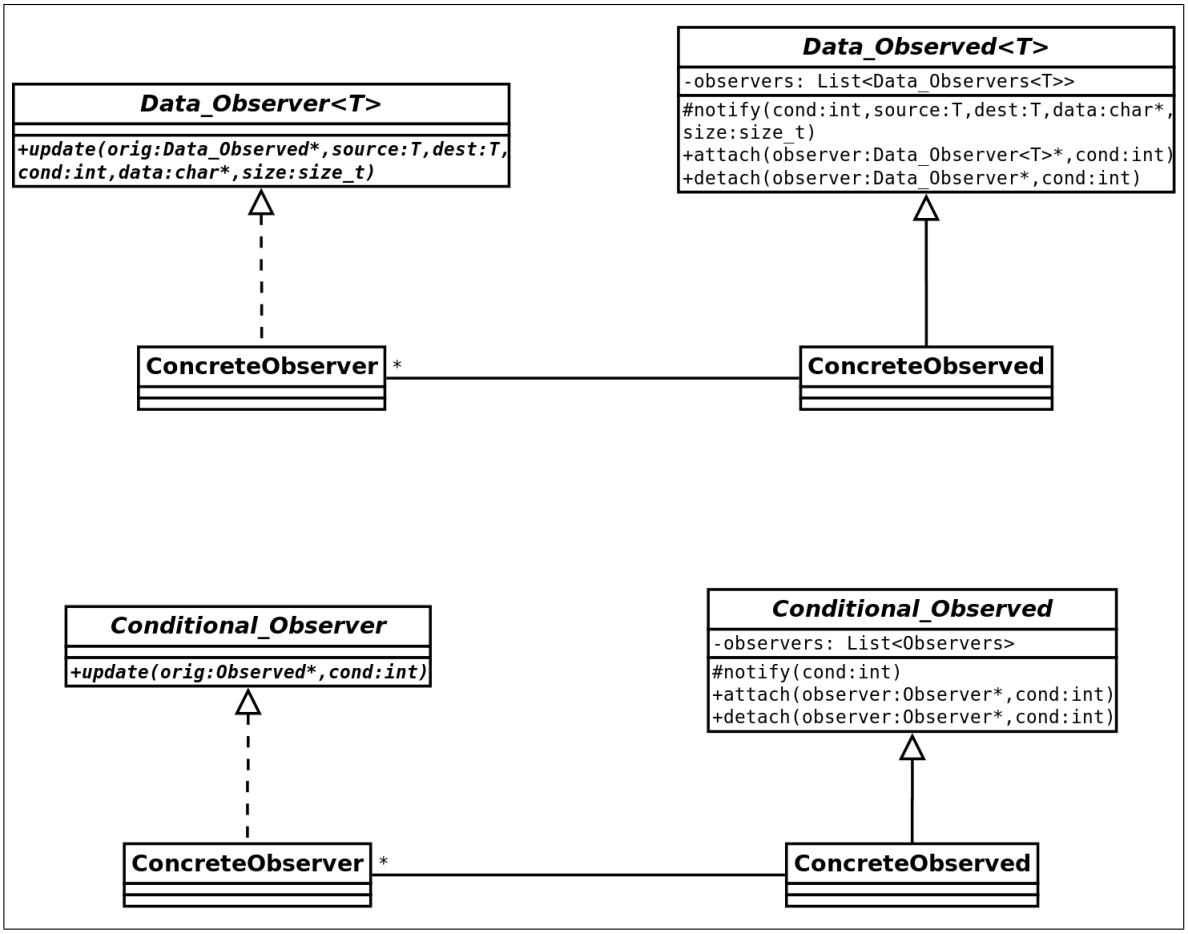

Figura 3.2: Diagrama das classes relacionadas ao padrão Publish/Subscribe

3.1.2 Descrição do modelo sequencial/síncrono

O modelo sequencial, visto na figura 3.3, foi projetado como uma evolução do modelo orientado a eventos. Na API sequencial temos duas novas classes, UDP::Channel e TCP::Channel. Tais classes implementam os *callbacks* necessários das classes TCP::Socket e UDP::Socket e proveem uma interface de programação linear que se assemelha às interfaces modernas de linguagens como Java e C#. Na classe UDP::Channel, por exemplo, temos no lugar do callback received() o método receive(), que quando executado, bloqueia até que uma mensagem seja rececebida. Já no caso do TCP::Channel, temos a vantagem que todo controle de timeout e retransmissões é feito sem nenhuma intervenção do desenvolvedor da aplicação. Como visto na parte inferior da figura 3.3, o modo de utilização por parte da aplicação deixa de ser através do mecanismo de herança e passa a ser através de associação/composição.

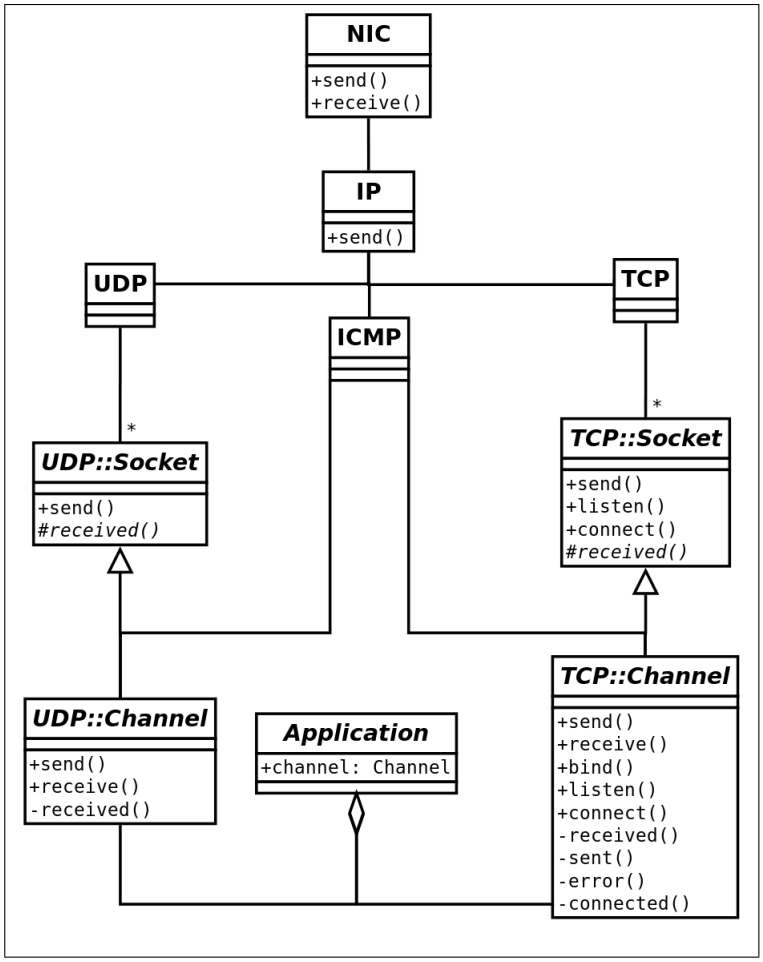

Figura 3.3: Diagrama de classes simples do modelo síncrono

Vantagens e desvantegens

O modelo sequencial é o mais simples de ser utilizado, possuindo semântica equivalente a tradicional API de *Sockets* do BSD, difundida entre os demais sistemas operacionais. A utilização de memória, tanto para código, quanto em tempo de execução, porém, é maior neste caso. O que pode, em algumas circunstancias inviabilizar sua utilização.

3.2 Comportamento

O objetivo dessa seção é analisar melhor a implementação de cada aspecto das diversas camadas de software que compõe este trabalho. Como houve desde o início

a preocupação entre relacionar diretamente as camadas do modelo TCP/IP diretamente com classes em um modelo orientado a objetos, também houve a preocupação em fazer com que cada classe (ou camada) utilize diretamente a camada inferior e forneça serviços à camada superior, porém sem depender desta, assim como proposto no modelo de referência OSI. Tal aproximação entre uma implementação prática e o modelo teórico permite a análise separada de cada componente da implementação, facilitando a depuração de eventuais problemas e até mesmo o estudo para fins didáticos.

3.2.1 Camada de rede

A camada central deste projeto é a camada de rede, representada pela classe IP. De uma maneira geral, podemos dizer que cada objeto IP funciona como um ponto central de coordenação. O padrão de projetos *Active Object* foi escolhido como base para a classe IP pelos seguintes motivos:

1. O EPOS possui uma implementação de *threading* sofisticada, logo o confinamento do processamento de pacotes IP dentro de uma *thread* permite que políticas de escalonamento sejam aplicadas. Se todo processamento necessário para recepção de pacotes IP fosse feito dentro de um contexto de tratamento de interrupções, a utilização de IP junto a um sistema de tempo real se tornaria quase impossível.

2. Nem toda interface de rede é dotada de mecanismos de sinalização assíncronos, como as interrupções. Logo a necessidade de utilizar o mecanismo de *polling* faz com que o recebimento de pacotes seja uma tarefa ativa e não uma ISR (*Interrupt Service Routine*).

As duas principais tarefas da camada IP são descritas de uma maneira simples pelos diagramas de atividade das figuras 3.4 e 3.5. O comportamento básico da recepção consiste em verificar consistências e notificar a camada superior (utilizando

o padrão Publish/Subscribe mencionado anteriormente). Já no envio de pacotes, a camada IP é responsável principalmente pelo encapsulamento correto dos dados (incluindo *checksums*). A descoberta de endereços físicos a partir de endereços lógicos é feita separadamente por uma classe chamada Network_Service. Dependendo do cenário de execução e das configurações que o desenvolvedor escolher para o sistema o Network_Service será implementado por uma classe distinta. Atualmente temos três Network_Service's distintos:

1. *ARP_Service:* Resolve endereços físicos utilizando o protocolo ARP. Útil para redes Ethernet.

2. *BCast_Service:* Envia todos os pacotes para o endereço de *broadcast*. Útil para conexões ponto-a-ponto e redes sem-frio.

3. *ADHOP_Service:* Utiliza o protocolo ADHOP (OKAZAKI; FRÖHLICH, 2009; OKAZAKI; FRÖHLICH, 2011) para determinar o próximo salto. Útil para redes sem-fio multi-hop.

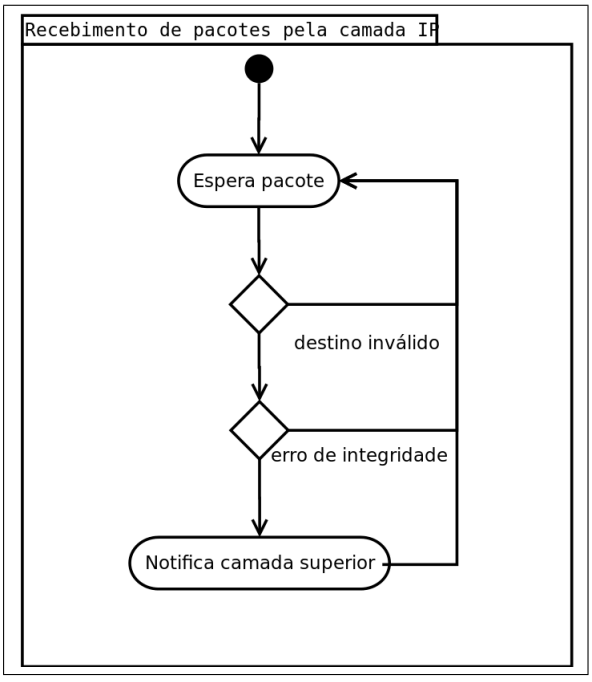

Figura 3.4: Diagrama de atividade do recebimento de pacotes pela camada de rede.

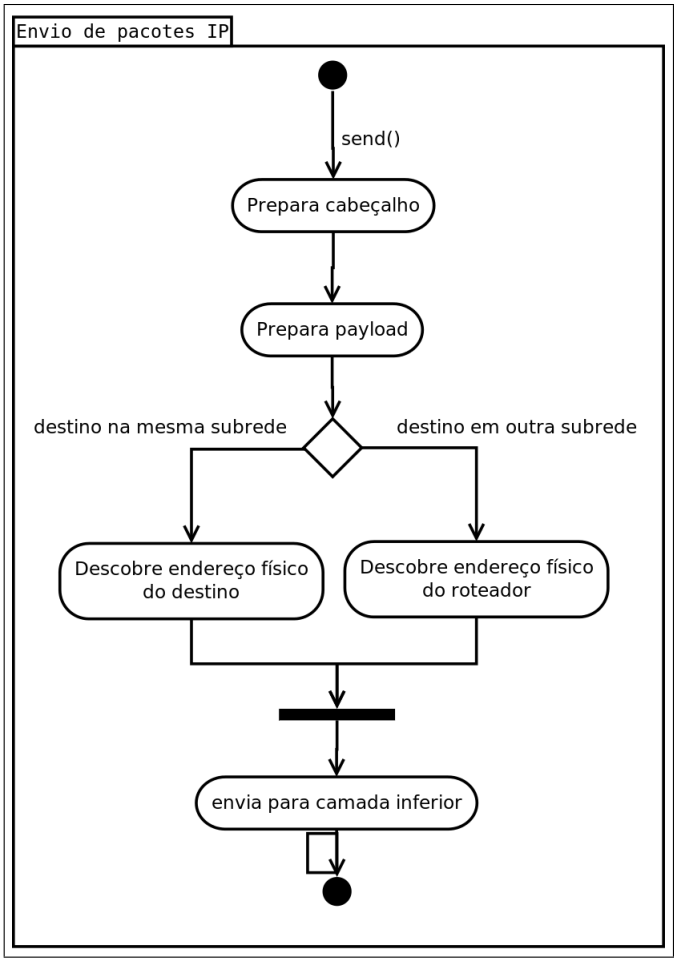

Figura 3.5: Diagrama de atividade do envio de pacotes pela camada de rede.

3.2.2 Camada de transporte: UDP

Atualmente apenas dois protocolos da camada de transporte são largamente utilizados, sendo eles o UDP e o TCP. Outros protocolos como o SCTP e DCCP (ONG; YOAKUM, 2002; KENT; ATKINSON, 1998) não são difundidos e por isso não foram contemplados neste trabalho.

Comparado ao TCP, o UDP é extremamente simples e por isso sua interface de software também é bastante reduzida. O principal recurso oferecido pelo UDP é a multiplexação, ou endereçamento a nível de aplicação, que permite que vários agentes (aplicações por exemplo) utilizem o mesmo endereço de rede para fluxos de comunicação distintos, sem que cada agente precise ter conhecimento da existência dos demais utilizadores da rede. De uma maneira geral, o UDP é amplamente utilizado

nos serviços de DNS, DHCP e transmissões multimídia. Cada um destes três tipos de serviço faz uso de uma funcionalidade (ou falta de) do UDP.

Para o DNS, a falta de estabelecimento de conexão é ideal, já que a consulta a nomes costuma ser composta de apenas duas mensagens distintas, uma de consulta e outra de resposta, assim sendo o estabelecimento e finalização de uma conexão, dobrariam a quantidade mínima necessária de troca de mensagens.

O DHCP, por sua vez, utilizado para configuração automática de atributos de um *host* dentro de uma rede, faz uso da capacidade do UDP de permitir endereçamento do tipo um-para-muitos (*broadcast*). Tal endereçamento se faz necessário pois no momento da autoconfiguração o *host* não possui nenhuma informação da rede e por isso não pode endereçar nenhum destino com precisão.

Os serviços multimídia, principalmente os de tempo-real ou baixa latência, tiram proveito da ausência de confirmação e retransmissão inerente ao UDP. Para a percepção humana, a perda de alguma informação de áudio/vídeo é menos importante ao entendimento do que o atraso ou a variação do mesmo (*jitter*).

Nossa implementação do UDP foi planejada de maneira a contemplar os cenários de utilização mencionados acima, com foco central em praticidade para o desenvolvedor. As figuras 3.6 e 3.7 esquematizam a estrutura das classes utilizas para comunicação UDP nos dois modelos desenvolvidos neste trabalho. Vale lembrar que algumas operações, como por exemplo a construção de instancias de UDP::Address a partir de representações textuais foram omitidas do diagrama para melhor entendimento.

3.2.3 Camada de transporte: TCP

3.2.4 Alterações no TCP

Neste trabalho algumas características ou funcionalidades originais do TCP não foram implementadas por motivos técnicos. Na sequência serão expostos tais itens e

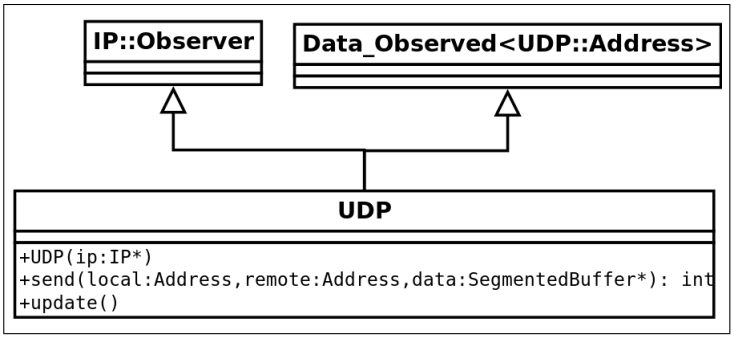

Figura 3.6: Esqueleto da classe UDP.

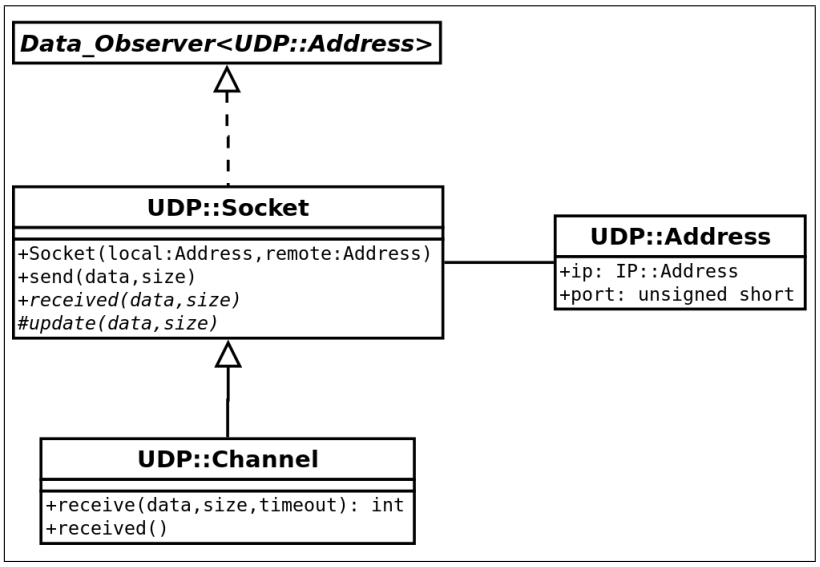

Figura 3.7: Interface de programação UDP::Socket e UDP::Channel.

o porquê da omissão de cada um.

Urgent Pointer A recomendação feita na seção 5 de Gont e Yourtchenko (2011) é para que novas aplicações não utilizem o mecanismo de Urgent Point do TCP. Os principais motivos são os problemas semânticos decorridos da implementação independente por diversos fabricantes. Na prática observa-se que praticamente nenhuma aplicação depende do TCP Urgent Point para o correto funcionamento. Logo, ignorar o mecanismo em questão não implica em nenhuma perda de utilidade para os cenários típicos em que esta implementação será utilizada.

Recebimento de dados no estado CLOSE_WAIT A operação **CLOSE** tradicionalmente implementada fecha apenas uma das pontas da conexão, permitindo ainda o recebimento de dados enquanto a outra ponta não executa a mesma operação. Para simplificar a semântica e manter um interface de programação mais simples, o método **close** da classe TCP::Channel foi projetado de modo a anunciar uma janela de tamanho zero e esperar o encerramento síncrono da conexão, não permitindo a utilização do método **receive** após seu uso. Na implementação orientada a eventos tal limitação não foi necessária, alem disso a classe TCP::Socket possui um callback chamado **closing** que pode ser utilizado para saber que a outra ponta fechou seu fluxo.

Controle de congestionamento Tal como no uIP, o controle de congestionamento desta implementação foi praticamente eliminado. As técnicas aplicadas no TCP-Reno e seus sucessores têm por objetivo a saturação da rede para melhor aproveitamento da banda e encaram perdas de pacote como sinal de congestionamento. Tal propriedade não é válida para todas as topologias de rede, como o IEEE 802.11.4 ou redes PLC. Outra característica associada ao controle de congestionamento é o *Delayed Ack*, sendo que este não foi implementado pois sua utilização junto ao nosso modelo de janela zero, que será explicando adiante, reduziria ainda mais o aproveitamento da banda de transmissão.

3.2.5 TCP com “Janela Zero”

O controle de fluxo nas conexões TCP é feito através da janela de recebimento. A janela de recebimento é um campo presente em todos os seguimentos TCP que indica quantos bytes o remetente está disposto a receber. No uIP a janela anunciada é igual ao MSS, sendo que tal característica advém da natureza orientada a eventos do uIP e da utilização de um único buffer universal para envio e recebimento. Por outro lado, o tamanho da janela TCP em sistemas convencionais costuma ser alto e técnicas como a utilização de TCP-Options para aumentá-la além dos limites originais são utilizadas.

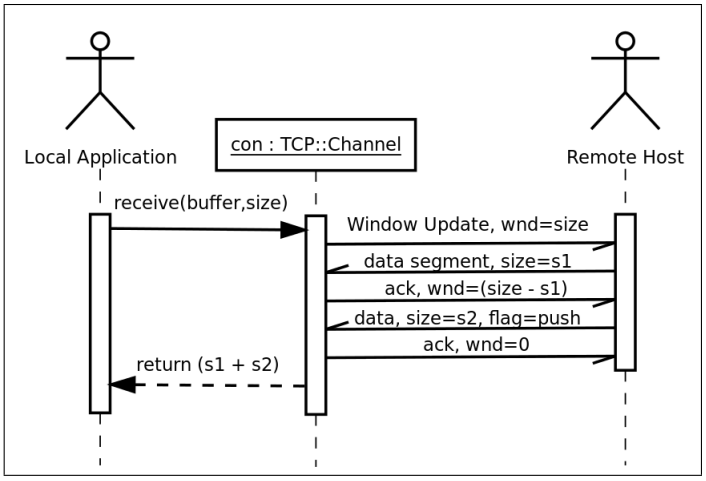

Figura 3.8: Recebimento de dados no TCP com janela zero

A quantidade de memória disponível em sistemas convencionais é várias ordens de magnitude superior ao necessário para o aproveitamento máxima da largura de banda pelo TCP e logo limitar o tamanho da janela se torna dispensável.

Levando os dois exemplos acima em consideração, nossa arquitetura abraça as duas propostas. Ao utilizar a API orientada a eventos a janela anunciada é sempre de um segmento, enquanto ao utilizar a API síncrona a janela anunciada passar a representar o buffer de recebimento fornecido pela aplicação, tal como representado no diagrama 3.8. O comportamento final, o qual damos o nome de TCP com Janela Zero, passa a ser caracterizado por dois estados:

- Um estado **idle** com janela de tamanho zero, quando a aplicação não está disposta a receber dados,
- Um estado **ativo** quando a aplicação executa o método **receive** em um Channel e a janela anunciada passa a ser o buffer fornecido pela aplicação.

Na operação de envio, a aplicação também possui um papel central no comportamento da camada de transporte. Novamente nossa implementação se equilibra entre a simplicidade do uIP e a eficiência dos grandes sistemas operacionais. No caso o uIP, todas as retransmissões são responsabilidades do desenvolvedor das aplicações, cabendo ao uIP apenas alertá-lo sobre os *timeouts* ocorridos. Em sistemas conven-

cionais, o nível de abstração é alto e o desenvolvedor fica completamente isolado de detalhes como retransmissão. Novamente, cada uma das duas APIs desenvolvidas se aproxima dos dois lados da questão:

- Utilizando TCP::Socket a aplicação pode utilizar deliberadamente a operação **send**, que retorna imediatamente. A cada segmento confirmado, o callback **sent** é executado para informar a aplicação quantos bytes a outra ponta recebeu. Em caso de timeout, o callback **error** com o parâmetro ERR_TIMEOUT é utilizado para informar a aplicação do problema. Neste cenário a aplicação decide como proceder com a retransmissão.

- A classe TCP::Channel, por outro lado, fornece uma operação **send** que bloqueia até que todos os dados tenham seu recebimento confirmado ou algum erro aconteça. Neste caso a aplicação fica isolada do problema das retransmissões, porém a utilização da banda depende diretamente do tamanho dos dados fornecido pela aplicação, quanto maior o buffer, maior o aproveitamento.

3.3 Configurabilidade

4 Resultados

Como visto nas tabelas 4.1 e 4.2, a diferença de tamanho entre o código gerado é mínima para a arquitetura Intel x86, com uma uma interface de rede Ethernet e utilizando ARP, e para a arquitetura ARM, no caso o EPOS Mote II, com C-MAC em uma rede 802.15.4 e roteamento ADHOP. A principal diferença encontrada no tamanho do componente IP, que ficou maior no ARM, deve-se ao fato da classe IP absorver o código do C-MAC e do ADHOP, que por sua vez são mais complexos que o código necessário para suportar Ethernet usando ARP. Vale a pena ressaltar que nem todo código gerado será transportado para o binário final da aplicação, isto se deve ao fato de aplicarmos técnicas de otimização do GCC durante a compilação e a link-edição, que fazem com que a imagem final das aplicações possua apenas as funções necessárias para seu funcionamento. Tais técnicas são conhecidas como *whole-program optmizations*.

Componente	**Tamanho (bytes)**
IP	12376
ICMP	3043
UDP	5331
TCP	13334
DHCP	1748

Tabela 4.1: Tamanho total dos arquivos-objeto de cada componente para arquitetura Intel x86 utilizando Ethernet e ARP.

A análise de algumas aplicações simples permite ter uma melhor ideia do espaço total ocupado, já que cada aplicação carrega apenas a funcionalidade necessária para desempenhar sua função. A tabela 4.3 mostra o tamanho total de três aplicativos de

Componente	Tamanho (bytes)
IP	12224
ICMP	2364
UDP	4920
TCP	14364
DHCP	1692

Tabela 4.2: Tamanho total dos arquivos-objeto de cada componente para arquitetura ARM utilizando ADHOP e CMAC.

teste, utilizando protocolos convencionais e interoperando com outros sistemas. O aplicativo Ping/Pong realiza requisições de ICMP ECHO e dá respostas ICMP ECHO-REPLY. O cliente DHCP faz auto-configuração de rede através do protocolo DHCP, que por sua vez foi implementado com UDP::Socket. A diferença entre o cliente WEB 1 e 2 é que o primeiro utiliza TCP::Socket e o segundo TCP::Channel.

Aplicação	Tamanho x86 (bytes)	Tamanho ARM (bytes)
Ping/Pong	27652	38260
Cliente DHCP	31044	41084
Cliente WEB 1	36368	47676
Cliente WEB 2	38768	49452

Tabela 4.3: Tamanho total dos arquivos-objeto de cada aplicativo de teste.

Classe	Memória RAM ocupada (bytes)
IP	184
ICMP	32
UDP	32
TCP	32
UDP::Socket	28
TCP::Socket	108
UDP::Channel	76
TCP::Socket	172

Tabela 4.4: Memória ocupada por cada instancia das classes da hierarquia de protocolos.

Na tabela 4.4 encontra-se o *footprint* de memória de cada objeto instanciado das classes da arquitetura TCP/IP. Como nossa arquitetura não impõe nenhum limite em

tempo de compilação na quantidade máxima de conexões abertas, tal informação pode ser usada para o desenvolvedor impor algum limite em tempo de execução e planejar a capacidade do sistema.

5 Conclusão

Neste trabalho foi apresentada a pilha de protocolos TCP/IP desenvolvida para o sistema EPOS. Tal implementação é de grande utilidade para compatibilidade com sistemas legados e interoperabilidade com sistemas diversos. Os resultados obtidos mostram a viabilidade de utilização na atual plataforma EPOSMote2, projetada para atender demandas acadêmicas e industriais nas mais diversas áreas de sistemas profundamente embarcados. Atualmente já há um trabalho em desenvolvimento utilizando esta pilha de protocolos como base para comunicação SIP/RTP e IEEE1451, o que ajudará a consolidar os artefatos desenvolvidos neste trabalho.

Por fim, podemos dizer que a ideia central do título de criar uma pilha flexível, configurável e de baixo custo para sistemas embarcados foi alcançada pelo desenvolvimento de uma elaborada hierarquia que permite ao desenvolvedor usar apenas o necessário para alcançar seus objetivos, ou seja, flexível e configurável. A utilização de algoritmos e estruturas de dados simples, resultando em pouco código, pouca utilização de CPU e RAM, confere o toque final de baixo custo ao trabalho.

5.1 Trabalhos futuros

Devida a limitação natural de tempo e recursos humanos de um TCC, foi necessário restringir o escopo do trabalho à apenas uma base conceitual e de software suficiente para garantir sua utilidade e permitir expansão futura. A continuidade natural deste trabalho será o desenvolvimento do suporte a IPv6 e a camada de adaptação

6lowPAN para integração de IPv6 em redes IEEE 802.15.4. A arquitetura desenvolvida foi propositalmente desacoplada para permitir a evolução da camada de rede de forma transparente à camada de transporte, diferentemente de outras implementações para sistemas embarcados e com a possibilidade de um crescimento incremental da complexidade do software envolvido.

Referências Bibliográficas

ABDELJAOUAD, I. et al. Performance analysis of modern TCP variants: A comparison of Cubic, Compound and New Reno. *25th Biennial Symposium on Communications (QBSC)*, Kingston, ON, USA, p. 80–83, May 2010. ISSN 978-1-4244-5709-0.

DUNKELS, A. Full TCP/IP for 8 Bit Architectures. In: *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*. San Francisco: [s.n.], 2003. Disponível em: <http://www.sics.se-/~adam/mobisys2003.pdf>.

DUNKELS, A. *Contiki: Bringing IP to Sensor Networks*. jan. 2009. ERCIM News. Disponível em: <http://ercim-news.ercim.org/content/view/496/705/>.

DUNKELS, A.; GRÖNVALL, B.; VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In: *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*. Tampa, Florida, USA: [s.n.], 2004. Disponível em: <http://www.sics.se/~adam/dunkels04contiki.pdf>.

DUNKELS, A.; SCHMIDT, O. *Protothreads - Lightweight Stackless Threads in C*. [S.l.], mar. 2005. Disponível em: <http://www.sics.se/~adam/dunkels05protothreads.pdf>.

DUNKELS, A. et al. Connecting Wireless Sensornets with TCP/IP Networks. In: *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*. Frankfurt (Oder), Germany: [s.n.], 2004. (C) Copyright 2004 Springer Verlag. http://www.springer.de/comp/lncs/index.html. Disponível em: <http://www.sics.se/~adam/wwic2004.pdf>.

EUGSTER, P. T. et al. The many faces of publish/subscribe. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 35, p. 114–131, June 2003. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/857076.857078>.

FRÖHLICH, A. A. *Application-Oriented Operating Systems*. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001. 200 p. ISBN 3-88457-400-0.

GONT, F.; YOURTCHENKO, A. *On the Implementation of the TCP Urgent Mechanism*. IETF, jan. 2011. RFC 6093 (Proposed Standard). (Request for Comments, 6093). Disponível em: <http://www.ietf.org/rfc/rfc6093.txt>.

GOODE, B. Voice over internet protocol (voip). *Proceedings of the IEEE*, IEEE, v. 90, n. 9, p. 1495–1517, 2002. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03-/wrapper.htm?arnumber=1041060>.

HA, S.; RHEE, I.; XU, L. Cubic: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 42, p. 64–74, July 2008. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1400097.1400105>.

HOWES, T.; SMITH, M. *The LDAP Application Program Interface*. IETF, ago. 1995. RFC 1823 (Informational). (Request for Comments, 1823). Disponível em: <http://www.ietf.org/rfc/rfc1823.txt>.

KENT, S.; ATKINSON, R. *Security Architecture for the Internet Protocol*. IETF, nov. 1998. RFC 2401 (Proposed Standard). (Request for Comments, 2401). Obsoleted by RFC 4301, updated by RFC 3168. Disponível em: <http://www.ietf.org/rfc/rfc2401-.txt>.

LI, Y.-C.; CHIANG, M.-L. Lyranet: A zero-copy tcp/ip protocol stack for embedded operating systems. *Real-Time Computing Systems and Applications, International Workshop on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 123–128, 2005. ISSN 1533-2306.

MASCOLO, S. Modeling the internet congestion control using a smith controller with input shaping. *Control Engineering Practice*, v. 14, n. 4, p. 425 – 435, 2006. ISSN 0967-0661. Disponível em: <http://www.sciencedirect.com/science/article/pii-/S0967066105000572>.

MOCKAPETRIS, P. *Domain names - implementation and specification*. IETF, nov. 1987. RFC 1035 (Standard). (Request for Comments, 1035). Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966. Disponível em: <http://www.ietf.org/rfc/rfc1035.txt>.

OKAZAKI, A. M.; FRÖHLICH, A. A. Adapting HOPNET Algorithm for Wireless Sensor Networks. In: *International Information and Telecommunication Technologies Symposium*. Florianópolis, Brazil: [s.n.], 2009. p. 191–194. ISBN 978-85-89264-10-5.

OKAZAKI, A. M.; FRÖHLICH, A. A. AD-ZRP: Ant-based Routing Algorithm for Dynamic Wireless Sensor Networks. In: *18th International Conference on Telecommunications*. Ayia Napa, Cyprus: [s.n.], 2011. p. 15–20. ISBN 978-1-4577-0023-1.

ONG, L.; YOAKUM, J. *An Introduction to the Stream Control Transmission Protocol (SCTP)*. IETF, maio 2002. RFC 3286 (Informational). (Request for Comments, 3286). Disponível em: <http://www.ietf.org/rfc/rfc3286.txt>.

POLPETA, F. V.; FRÖHLICH, A. A. Hardware Mediators: a Portability Artifact for Component-Based Systems. In: *International Conference on Embedded and Ubiquitous Computing*. Aizu, Japan: Springer, 2004. (Lecture Notes in Computer Science, v. 3207), p. 271–280. ISBN 354022906x.

POSTEL, J. *User Datagram Protocol*. IETF, ago. 1980. RFC 768 (Standard). (Request for Comments, 768). Disponível em: <http://www.ietf.org/rfc/rfc768.txt>.

POSTEL, J. *Internet Protocol*. IETF, set. 1981. RFC 791 (Standard). (Request for Comments, 791). Updated by RFC 1349. Disponível em: <http://www.ietf.org/rfc-/rfc791.txt>.

POSTEL, J. *Transmission Control Protocol*. IETF, set. 1981. RFC 793 (Standard). (Request for Comments, 793). Updated by RFCs 1122, 3168, 6093. Disponível em: <http://www.ietf.org/rfc/rfc793.txt>.

SHELBY, Z. et al. NanoIP: The Zen of Embedded Networking. *PROCEEDINGS OF THE IEEE*, p. 1218–1222, 2003.

TAN, K.; SONG, J. Compound TCP: A scalable and TCP-friendly congestion control for high-speed networks. In: *in 4th International workshop on Protocols for Fast Long-Distance Networks (PFLDNet), 2006*. [S.l.: s.n.], 2006.

TANENBAUM, A. *Computer Networks*. 4th. ed. [S.l.]: Prentice Hall Professional Technical Reference, 2002. ISBN 0130661023.

VASSEUR, J.-P.; DUNKELS, A. *Interconnecting Smart Objects with IP - The Next Internet*. Morgan Kaufmann, 2010. ISBN 978-0123751652. Disponível em: <http://TheNextInternet.org/>.

Apêndice A – Infra-estrutura

A.1 Cabeçalhos

A.1.1 ip.h

```
#ifndef __ip_h
#define __ip_h

#include <active.h>
#include <nic.h>
#include <service.h>
#include <utility/malloc.h>
#include <utility/debug.h>
#include <utility/buffer.h>
#include <utility/string.h>
#include <system/meta.h>
#include <thread.h>

// Common aliases
typedef unsigned char u8;
typedef unsigned short u16;
typedef unsigned long u32;
typedef signed   long  s32;

__BEGIN_SYS

class IP_Address : public NIC_Common::Address<4> {
public:
    IP_Address() {}
    IP_Address(unsigned char addr[4])
        : NIC_Common::Address<4>(addr[0],addr[1],addr[2],addr[3]) {}

    IP_Address(unsigned long addr) {
        addr = CPU::htonl(addr);
        memcpy(this, &addr, sizeof(addr));
    }
    IP_Address(u8 a0, u8 a1 = 0, u8 a2 = 0, u8 a3 = 0)
        : NIC_Common::Address<4>(a0, a1, a2, a3) {}


    //* create from string representation in the form A.B.C.D
    IP_Address(const char * _addr);
```

```
    friend Debug& operator<<(Debug& db,const IP_Address& addr);

    //* convert to string pointed by dst and return last char position
    char* to_string(char * dst);

    bool is_neighbor(IP_Address other,IP_Address mask) const
    {
        u32 c1 = u32(*this) & u32(mask);
        u32 c2 = u32(other) & u32(mask);
        return c1 == c2;
    }

    operator u32() { return *reinterpret_cast<u32 *>(this); }
    operator u32() const { return *reinterpret_cast<const u32 *>(this); }
};

class IP : public Traits<IP>,
           public Active,
           public Data_Observed<IP_Address>
{
public:

    // Definitions

    typedef Data_Observer<IP_Address> Observer;
    typedef IP_Address                Address;
    typedef NIC::Address              MAC_Address;
    typedef u8                        Protocol;

    typedef
        Service<Traits<IP>::SERVICE>::Network_Service<NIC,IP>
        Network_Service;

    static const u16 MTU = ~0;
    static const Address NULL;

    /// This flag is used for single-NIC optmizations
    static const bool multiNIC = (Traits<NIC>::NICS::Length > 1);

    enum {
        PROT_IP = NIC::IP,
        PROT_ARP = NIC::ARP,
        PROT_RARP = NIC::RARP
    };

    class Header;

    // Methods

    const Address & address() { return _self; }
    const Address & gateway() { return _gateway; }
    const Address & netmask() { return _netmask; }

    void set_address(const Address & addr) { _self = addr; }
    void set_gateway(const Address & addr) { _gateway = addr; }
    void set_netmask(const Address & addr) { _netmask = addr; }

    s32 send(const Address & to,const char * data,u16 size,Protocol proto) {
```

```
        SegmentedBuffer sb(data,size);
        return send(_self,to,&sb,proto);
    }

    s32 send(const Address & to,SegmentedBuffer * data,Protocol proto) {
        return send(_self,to,data,proto);
    }

    s32 send(const Address & from,const Address & to,SegmentedBuffer * data,Protocol proto);

    IP(unsigned int unit=0);
    ~IP();

//  void update(NIC::Observed * o, int p);

    int run();
    void process_incoming();
    void process_ip(char * data,u16 size);

    NIC * nic() { return &_nic; }
    const MAC_Address & hw_address() { return _nic.address(); }
    const int hw_address_len() { return sizeof(MAC_Address); }
    const u16 mtu() { return _nic.mtu(); }

    static u16 calculate_checksum(const void* ptr, u16 count);

    static IP* instance(unsigned int unit = 0) {
        if (!_instance[unit])
            _instance[unit] = new IP(unit);
        return _instance[unit];
    }

private:
    NIC _nic;
    char * _packet_receive[MAX_FRAGMENTS];
    char * _packet_send;
    u16    _packet_size[MAX_FRAGMENTS];
    int    _packet_count;

    Network_Service _network_service;

    Address _self;
    Address _netmask;
    Address _gateway;
    Address _broadcast;

    volatile bool _alive;
    Thread * _thread;

    static const unsigned int DEF_VER = 4;
    static const unsigned int DEF_IHL = 5; // 20 bytes
    static const unsigned int DEF_TOS = 0;
    static const unsigned int DEF_TTL = Traits<IP>::DEF_TTL;

    // Pseudo header for checksum calculations
    struct Pseudo_Header {
        u32 src_ip;
        u32 dst_ip;
```

```
        u8 zero;
        u8 protocol;
        u16 length;
    };

    static IP* _instance[Traits<NIC>::NICS::Length];
};

class IP::Header : public Traits<IP> {
public:
    enum {
        MF_FLAG = 1, // More Fragments
        DF_FLAG = 2 // Don't Fragment
    };

    Header() {}

    Header(const Address & src, const Address & dst, const Protocol & prot, u16 size) :
        _ihl(DEF_IHL + Traits<IP>::OPT_SIZE), _version(DEF_VER), _tos(DEF_TOS),
        _length(CPU::htons(sizeof(Header) + size)), _id(CPU::htons(pktid++)),
        _offset(0), _flags(0), _ttl(DEF_TTL), _protocol(prot), _checksum(0),
        _src_ip(src), _dst_ip(dst)
    {
        calculate_checksum();
    }

    const Address & src_ip() const { return _src_ip; }
    const Address & dst_ip() const { return _dst_ip; }
    u32 hlength() { return _ihl * 4; }
    u32 length() const { return CPU::ntohs(_length); }
    u16 flags() const {
        return CPU::ntohs(_flags << 13 | _offset) >> 13;
    }
    u16 offset() const {
        return CPU::ntohs(_flags << 13 | _offset) & 0x1fff;
    }
    const Protocol & protocol() const { return _protocol; }
    u16 id() const { return CPU::ntohs(_id); }

    // setters for fragment operations
    void set_src(const Address & src_ip){ _src_ip = src_ip; }
    void set_length(u16 length) { _length = CPU::htons(length); }
    void set_offset(u16 off) {
        u16 x = CPU::htons(flags()<<13|off);
        _offset = x&0x1fff;
        _flags  = x>>13;
    }
    void set_flags(u16 flg) {
        u16 x = CPU::htons(flg<<13|offset());
        _offset = x&0x1fff;
        _flags  = x>>13;
    }
    void set_protocol(u8 protocol){
        _protocol = protocol;
    }

    char* get_options() { return _opt; }
```

```
    u8 ttl() { return _ttl; }

    void ttl(u8 nttl) { _ttl = nttl; }

    void calculate_checksum();

    friend Debug& operator<< (Debug &db, const IP::Header &h) {
        IP::Address ip_src(h._src_ip), ip_dst(h._dst_ip);                          220
        u16 flags = h.flags();

        db << "{ver=" << h._version
        << ",ihl=" << h._ihl
        << ",tos=" << h._tos
        << ",len=" << CPU::ntohs(h._length)
        << ",id=" << CPU::ntohs(h._id)
        << ",off=" << h.offset()
        << ",flg=" << (flags == IP::Header::DF_FLAG ? "[DF]" :
        (flags == IP::Header::MF_FLAG ? "[MF]" : "[ ]"))                           230
        << ",ttl=" << h._ttl
        << ",pro=" << h._protocol
        << ",chk=" << (void *)h._checksum
        << ",src=" << ip_src
        << ",dst=" << ip_dst
        << "}";
        return db;
    }

private:                                                                           240
    u8  _ihl:4;       // IP Header Length (in 32-bit words)
    u8  _version:4; // IP Version
    u8  _tos;         // Type Of Service (no used -> 0)
    u16 _length;      // Size of datagram (header + data)
    u16 _id;          // Datagram id
    u16 _offset:13; // Fragment offset (x 8 bytes)
    u16 _flags:3;     // Flags (UN, DF, MF)
    u8  _ttl;         // Time To Live
    Protocol  _protocol;  // RFC 1700 (1->ICMP, 6->TCP, 17->UDP)
    volatile u16 _checksum; // Header checksum                                     250
    Address _src_ip;    // Source IP address
    Address _dst_ip;    // Destination IP addres
    char _opt[(4 * OPT_SIZE)];
    static unsigned short pktid;
};

__END_SYS

#endif
                                                                                   260
```

A.1.2 icmp.h

```
#ifndef ICMP_H
#define ICMP_H

#include <cpu.h>
```

```
#include <ip.h>

__BEGIN_SYS

class ICMP_SingleNIC {
public:
    static ICMP * instance();
protected:
    inline ICMP_SingleNIC(IP * ip) ;

    IP * ip() const { return IP::instance(); }
private:
    static ICMP * _instance;
};

class ICMP_MultiNIC {
public:
    static ICMP * instance(unsigned int i=0);
protected:
    ICMP_MultiNIC(IP * ip) : _ip(ip) {}

    IP * ip() const { return _ip; }
private:
    IP * _ip;
    static ICMP * _instance[Traits<NIC>::NICS::Length];
};

class ICMP : public IF<IP::multiNIC, ICMP_MultiNIC, ICMP_SingleNIC>::Result,
             public IP::Observer,
             public Data_Observed<IP::Address> {
public:
    typedef Data_Observer<IP::Address> Observer;
    typedef Data_Observed<IP::Address> Observed;

    typedef IF<IP::multiNIC, ICMP_MultiNIC, ICMP_SingleNIC>::Result Base;

    static const unsigned short ICMP_ID = 1; // IP sub-protocol identifier

    typedef unsigned char Code;
    typedef unsigned char Type;

    enum /*Types*/{
        ECHO_REPLY = 0,
        UNREACHABLE = 3,
        SOURCE_QUENCH = 4,
        REDIRECT = 5,
        ALTERNATE_ADDRESS = 6,
        ECHO = 8,
        ROUTER_ADVERT = 9,
        ROUTER_SOLIC = 10,
        TIME_EXCEEDED = 11,
        PARAMETER_PROBLEM = 12,
        TIMESTAMP = 13,
        TIMESTAMP_REPLY = 14,
        INFO_REQUEST = 15,
        INFO_REPLY = 16,
        ADDRESS_MASK_REQ = 17,
        ADDRESS_MASK_REP = 18,
```

```
    TRACEROUTE = 30,
    DGRAM_ERROR = 31,
    MOBILE_HOST_REDIR = 32,
    IPv6_WHERE_ARE_YOU = 33,
    IPv6_I_AM_HERE = 34,
    MOBILE_REG_REQ = 35,
    MOBILE_REG_REP = 36,
    DOMAIN_NAME_REQ = 37,
    DOMAIN_NAME_REP = 38,
    SKIP = 39
};

enum /*Unreacheble Codes*/ {
    NETWORK_UNREACHABLE = 0,
    HOST_UNREACHABLE = 1,
    PROTOCOL_UNREACHABLE = 2,
    PORT_UNREACHABLE = 3,
    FRAGMENTATION_NEEDED = 4,
    ROUTE_FAILED = 5,
    NETWORK_UNKNOWN = 6,
    HOST_UNKNOWN = 7,
    HOST_ISOLATED = 8,
    NETWORK_PROHIBITED = 9,
    HOST_PROHIBITED = 10,
    NETWORK_TOS_UNREACH = 11,
    HOST_TOS_UNREACH = 12,
    ADMIN_PROHIBITED = 13,
    PRECEDENCE_VIOLATION = 14,
    PRECEDENCE_CUTOFF = 15
};

class Packet {
    friend class ICMP;
protected:
    unsigned char _type;
    unsigned char _code;
    unsigned short _checksum;
    unsigned short _id;
    unsigned short _sequence;
    char _data[56];
public:
    Packet(Type type,Code code,
           unsigned short id,unsigned short seq,
           const char * data = 0,short size = 56);

    const char* raw() {
        return reinterpret_cast<const char*>(this);
    }

    Type type() { return _type; }
    Code code() { return _code; }
    unsigned short id() { return CPU::htons(_id); }
    unsigned short sequence() { return CPU::htons(_sequence); }
    unsigned short checksum() { return _checksum; }

    char * data() { return _data; }
};
```

```
    ICMP(IP* ip = 0);
    ~ICMP();

    void update(Data_Observed<IP::Address> *ob, long c, IP::Address src,
            IP::Address dst, void *data, unsigned int size);

    void send(IP::Address to,Packet & pkt) {
        send(ip()->address(),to,pkt);
    }

    void send(IP::Address from,IP::Address to,Packet& pkt);

};

// This is here because we cannot use static_cast before ICMP is defined
ICMP_SingleNIC::ICMP_SingleNIC(IP * ip) {
    _instance = (ICMP*)(this);
}

__END_SYS

#endif // ICMP_H
```

A.1.3 dhcp.h

```
#ifndef DHCPC_H
#define DHCPC_H

#include "udp.h"
#include <utility/random.h>

__BEGIN_SYS
/*
 * Reference:
 * DHCP:         http://www.ietf.org/rfc/rfc2131.txt
 * DHCP options: http://www.ietf.org/rfc/rfc2132.txt
 */
class DHCP {
public:
    // Limited to opt_size = 308

    template<int opt_size> class Packet {
    public:
        u8 _op, _htype, _hlen, _hopts;
        u32 _xid;
        u16 _secs, _flags;
        u32 _ciaddr, _yiaddr, _siaddr, _giaddr;
        u8 _chaddr[16];
        u8 _sname[64];
        u8 _file[128];
        u8 _magic[4];
        u8 _options[opt_size];
        u8 _end;
        u8 _padding[312 - 5 - opt_size];
```

```
        u8 op() const { return _op; }
        u32 xid() const { return _xid; }
        u16 secs() const { return CPU::ntohs(_secs); }
        u32 your_address() const { return CPU::ntohl(_yiaddr); }
        u32 server_address() const { return CPU::ntohl(_siaddr); }
        u8 * options() const { return const_cast<u8 * const>(_options); }

        Packet()
        {
            memset(&_op, 0, sizeof(Packet));
            _magic[0] = 99; // magic cookie
            _magic[1] = 130;
            _magic[2] = 83;
            _magic[3] = 99;
            _end = 255; // end of options
        }

    };

    class Discover : public Packet<3> {
    public:
        Discover(IP * _net) : Packet<3>() {
            _op = 1;
            _htype = 1;
            _hlen = _net->hw_address_len();
            _xid = Pseudo_Random::random();
            memcpy(_chaddr, &_net->hw_address(), _hlen);
            _options[0] = 53; // DHCPMSG
            _options[1] = 1;  // message size
            _options[2] = 1;  // dhcp discover
        }

    };

    class Request : public Packet<8> {
    public:
        Request(IP * _net,const Packet<255> * discovered) : Packet<8>() {
            _op = 1;
            _htype = 1;
            _hlen = _net->hw_address_len();
            _xid = discovered->_xid;
            _ciaddr = discovered->_ciaddr;
            _siaddr = discovered->_siaddr;
            memcpy(_chaddr, &_net->hw_address(), _hlen);
            _options[0] = 53; // DHCP message
            _options[1] = 1;  // size
            _options[2] = 3;  // dhcp discover
            _options[3] = 55; // parameter request
            _options[4] = 3;  // size
            _options[5] = 1;  // subnet
            _options[6] = 3;  // router
            _options[7] = 6;  // dns
        }
    };

    class Client;
};
```

```
class DHCP::Client : public UDP::Socket {
public:
    enum {
        IDLE,
        DISCOVER,
        REQUEST,
        RENEW,
        RELEASE
    };

    Client(UDP * udp = 0);

    ~Client() {}

    void received(const UDP::Address & src,const char *data, unsigned int size);

    void configure();
    void parse_options(const Packet<255> * packet);
    void renew();
    void release();

    IP::Address address() { return _ip; }
    IP::Address netmask() { return _mask; }
    IP::Address gateway() { return _gw; }
    IP::Address brodcast() { return _bcast; }
    IP::Address nameserver() { return _ns; }

protected:
    short _state;
    u32 _xid;
    u32 _lease_time;
    IP::Address _ip, _mask, _gw, _bcast, _ns;
};

__END_SYS

#endif
```

A.1.4 udp.h

```
#ifndef __udp_h
#define __udp_h

#include <alarm.h>
#include <ip.h>
#include <icmp.h>
#include <utility/handler.h>
#include <mutex.h>

__BEGIN_SYS

class UDP_Address {
public:
    /// Creates a NULL address.
```

```
    UDP_Address() { }

    /// Creates an UDP_Address from an ip number and port.
    UDP_Address(u32 ip, u16 port): _ip(ip), _port(port) { }

    /// Creates and UDP_Address from an IP::Address object and port number.
    UDP_Address(IP_Address ip, u16 port): _ip(ip), _port(port) { }

    /**
     * Creates and UDP_Address from its string representation in the
     * a.b.c.d:port format.
     */
    UDP_Address(const char *addr);


    /// Output the UDP_Address string in the format a.b.c.d:port to a stream.
    template < typename T >
    friend T & operator <<(T & out, const UDP_Address & h) {
        out << dec << h.ip() << ":" << (unsigned int)(h.port());
        return out;
    }

    /// Write the string representation in the format a.b.c.d:port to _dst_.
    char* to_string(char * dst);

    /// Get port number.
    u16 port() const { return _port; }

    /// Get IP address.
    IP_Address ip() const { return _ip; }

    /**
     * Change port number, beware that a Channel using this Address
     * will not be be bound to this port unless you use UDP::Channel->local().
     */
    void port(u16 new_port) { _port = new_port; }

    /**
     * Change IP address for this UDP_Address, beware that a Channel
     * using this Address will not be be bound to this IP unless you
     * use UDP::Channel->local().
     */
    void ip(const IP_Address& ip) { _ip = ip; }

    /// Compare this address to other and returns true if both are equal.
    bool operator==(const UDP_Address& other)
    {
        return ip() == other.ip() && port() == other.port();
    }

private:

    IP_Address _ip;
    u16 _port;
};

// Optmization for single NIC support
class UDP_SingleNIC {
```

```
public:
    IP* ip() const { return IP::instance(); }

protected:
    UDP_SingleNIC(IP * ip) {}
};

// Generalization for multiple NICs support
class UDP_MultiNIC {
public:
    IP* ip() const { return _ip; }

protected:
    UDP_MultiNIC(IP * ip) : _ip(ip) {}

private:
    IP * _ip;
};

class UDP : public IF<IP::multiNIC, UDP_MultiNIC, UDP_SingleNIC>::Result,
            public IP::Observer,
            public Data_Observed < UDP_Address > {
protected:
    typedef IF<IP::multiNIC, UDP_MultiNIC, UDP_SingleNIC>::Result Base;

public:
    // UDP ID (IP Frame)
    static const IP::Protocol ID_UDP = 0x11;

    typedef UDP_Address Address;

    class Header;
    class Socket;
    class Channel;

    class Socket_SingleNIC;
    class Socket_MultiNIC;

    class Channel_SingleNIC;
    class Channel_MultiNIC;

    UDP(IP * ip = 0);

    ~UDP();

    s32 send(Address local, Address remote, SegmentedBuffer * data);

    // Data_Observer callback
    void update(Data_Observed<IP::Address> *ob, long c, IP::Address src,
                IP::Address dst, void *data, unsigned int size);

    static UDP * instance(unsigned int i=0);

private:

    struct Pseudo_Header {
        u32 src_ip;
        u32 dst_ip;
```

```
        u8 zero;
        u8 protocol;
        u16 length;

        Pseudo_Header(u32 src,u32 dst,u16 len)
        : src_ip(src), dst_ip(dst), zero(0), protocol(ID_UDP),
          length(CPU::htons(len)) {};
    };
};

class UDP::Header {
    friend class UDP;
public:
    Header() {}

    Header(u16 src_port = 0, u16 dst_port = 0, u16 data_size = 0)
    : _src_port(CPU::htons(src_port)),
      _dst_port(CPU::htons(dst_port)),
      _length(CPU::htons(sizeof(UDP::Header) + data_size)),
      _checksum(0) { }

    void checksum(IP::Address src,IP::Address dst,SegmentedBuffer * sb);

    u16 dst_port() const { return CPU::ntohs(_dst_port); }
    u16 src_port() const { return CPU::ntohs(_src_port); }

    friend Debug & operator <<(Debug & db, const Header & h) {
        db << "{sprt=" << CPU::ntohs(h._src_port)
           << ",dprt=" << CPU::ntohs(h._dst_port)
           << ",len=" << CPU::ntohs(h._length)
           << ",chk=" << (void *)h._checksum << "}";
        return db;
    }

private:
    u16 _src_port;  // Source UDP port
    u16 _dst_port;  // Destination UDP port
    u16 _length;    // Length of datagram (header + data) in bytes
    volatile u16 _checksum; // Pseudo header checksum (see RFC)
};

// Socket optmization for single NIC scenario
class UDP::Socket_SingleNIC {
public:
    UDP* udp() const { return UDP::instance(); }
protected:
    Socket_SingleNIC(UDP * udp) {}
};

// Socket optmization for multiple NICs scenario
class UDP::Socket_MultiNIC {
public:
    UDP* udp() const { return _udp; }
protected:
    Socket_MultiNIC(UDP * udp) : _udp(udp) {
        if (!udp) _udp = UDP::instance();
    }
private:
```

```
    UDP * _udp;
};

class UDP::Socket : public IF<IP::multiNIC,
                              Socket_MultiNIC, Socket_SingleNIC>::Result,
                    public Data_Observer <UDP_Address>
{
    friend class UDP;
    typedef IF<IP::multiNIC,
               Socket_MultiNIC, Socket_SingleNIC>::Result Base;
public:
    Socket(Address local, Address remote, UDP * udp = 0);

    ~Socket();

    s32 send(const char *data, u16 size) const {
        SegmentedBuffer sb(data, size);
        return send(&sb);
    }
    s32 send(SegmentedBuffer * data) const {
        return udp()->send(_local, _remote, data);
    }

    void local(const Address & local) { _local = local; }
    void remote(const Address & party) { _remote = party; }

    const Address & remote() const { return _remote; }

    void update(Observed * o, long c, UDP_Address src, UDP_Address dst,
                void * data, unsigned int size);

    // every Socket should implement one
    virtual void received(const Address & src,
                          const char *data, unsigned int size) {};
protected:
    Address _local;
    Address _remote;
};

class UDP::Channel_SingleNIC {
public:
    ICMP * icmp() const { return ICMP::instance(); }
};

class UDP::Channel_MultiNIC {
public:
    ICMP * icmp() const { return _icmp; }
protected:
    Channel_MultiNIC();
private:
    ICMP * _icmp;
};

/**
 * The UDP::Channel is the top level class for using the UDP/IP protocol.
 * It has a simple send/receive interface with rendezvou semantics.
 */
class UDP::Channel : public Socket,
```

```
                    public IF<IP::multiNIC,
                              Channel_MultiNIC, Channel_SingleNIC>::Result,
                    public ICMP::Observer,
                    public Handler                                          250
{
public:
    /// Return conditions
    static const int DESTROYED = -1;
    static const int TIMEOUT = -2;


    /**
     * Return non-zero on error condition.
     * Error values are from ICMP::Packet unreachable codes.                 260
     */
    unsigned char error() { if (_error) return ~_error; }

    /**
     * Wait for incoming data. A max of _size_ bytes will be written to _buf_.
     * Returns the amount received or a negative value in case of error.
     * Blocks indefinitely.
     */
    int receive(Address * from,char * buf,unsigned int size);
                                                                            270
    /**
     * Wait for incoming data. A max of _size_ bytes will be written to _buf_.
     * Returns the amount received or a negative value in case of error.
     * Blocks for _timeout_ microseconds at max.
     */
    int receive(Address * from,char * buf,unsigned int size,
                const Alarm::Microsecond& timeout);

    /**
     * Sends _size_ bytes from _buf_ to destination _to_.                    280
     */
    //int send(const Address& to,const char * buf,unsigned int size);


    Channel(const Address& local,const Address& remote);

    /**
     * Destroys the channel. If a thread is blocked on a receive call it will
     * be released but the result can be catastrophic.
     */                                                                     290
    ~Channel();


protected:
    // Socket callback
    virtual void received(const Address & src,
                          const char *data, unsigned int size);

    // ICMP callback
    void update(Data_Observed<IP::Address> *ob, long c,                     300
                IP::Address src, IP::Address dst,
                void *data, unsigned int size);

    // Handler callback
```

```
    void operator()();

    Address * _buffer_src;
    Condition   _buffer_wait;
    unsigned int _buffer_size;
    char      * _buffer_data;
    unsigned char _error;
};


__END_SYS
#endif
```

A.1.5 tcp.h

```
// EPOS Transmission Control Protocol implementation

#ifndef __tcp_h
#define __tcp_h

#include <alarm.h>
#include <condition.h>
#include <ip.h>
#include <icmp.h>
#include <udp.h> // TCP::Address == UDP_Address
#include <utility/handler.h>
#include <utility/random.h>

__BEGIN_SYS

// Optmization for single NIC support
class TCP_SingleNIC {
public:
    IP* ip() const { return IP::instance(); }
protected:
    TCP_SingleNIC(IP * ip) {}
};

// Generalization for multiple NICs support
class TCP_MultiNIC {
public:
    IP* ip() const { return _ip; }

protected:
    TCP_MultiNIC(IP * ip) : _ip(ip) {}

private:
    IP * _ip;
};

class TCP: public IF<IP::multiNIC, TCP_MultiNIC, TCP_SingleNIC>::Result,
           public IP::Observer,
           public Data_Observed < UDP_Address >
{
public:
```

```
    typedef IF<IP::multiNIC, TCP_MultiNIC, TCP_SingleNIC>::Result Base;
    typedef UDP_Address Address;

    static const unsigned int ID_TCP = 6;

    class Header;

    class Socket_SingleNIC;
    class Socket_MultiNIC;

    class Socket;
    class ServerSocket;
    class ClientSocket;

    class Channel;

    TCP(IP * ip = 0);
    ~TCP();

    void update(Data_Observed<IP::Address>*, long, IP::Address,
                IP::Address, void*, unsigned int);

    static TCP * instance(unsigned int i=0);

    inline u16 mss();
};

class TCP::Header {
    friend class TCP;

 public:
    Header(u32 s = 0, u32 a = 0);

    u16 _src_port, _dst_port;
    u32 _seq_num, _ack_num;
#if !defined(i386) && !defined(AVR) && !defined(ARM)
    u8  _hdr_off:4, _un1:4;
    bool  _un3:1, _un2:1;
    bool  _urg:1, _ack:1, _psh:1, _rst:1, _syn:1, _fin:1;
#else
    u8  _un1:4, _hdr_off:4;
    bool  _fin:1, _syn:1, _rst:1, _psh:1, _ack:1, _urg:1;
    bool  _un2:1, _un3:1;
#endif
    u16 _wnd;
    volatile u16 _chksum;
    u16 _urgptr;

    // getters
    u16 size() const { return _hdr_off * 4; }
    u16 dst_port() const { return CPU::ntohs(_dst_port); }
    u16 src_port() const { return CPU::ntohs(_src_port); }
    u32 seq_num() const { return CPU::ntohl(_seq_num); }
    u32 ack_num() const { return CPU::ntohl(_ack_num); }
    u16 wnd() const { return CPU::ntohs(_wnd); }
    u16 chksum() const { return CPU::ntohs(_chksum); }
    u16 urgptr() const { return CPU::ntohs(_urgptr); }
```

```
    // setters
    void seq_num(u32 v) { _seq_num = CPU::htonl(v); }
    void ack_num(u32 v) { _ack_num = CPU::htonl(v); }
    void wnd(u16 v) { _wnd = CPU::htons(v); }
    void chksum(u16 v) { _chksum = CPU::htons(v); }
    void urgptr(u16 v) { _urgptr = CPU::htons(v); }
    void dst_port(u16 v) { _dst_port = CPU::htons(v); }
    void src_port(u16 v) { _src_port = CPU::htons(v); }

    u16 _checksum(IP::Address src,IP::Address dst,u16 len);
    void _checksum(IP::Address src,IP::Address dst,SegmentedBuffer * sb);
    void checksum(IP::Address src,IP::Address dst,u16 len);
    bool validate_checksum(IP::Address src,IP::Address dst,u16 len);

    // Ultimate unreadable coding style!
    friend Debug& operator<< (Debug & db, const Header& s)
    {
    db << "Header[SRC="<<s.src_port()<<",DST="<<s.dst_port()<<"] SEQ="<<
    s.seq_num()<<",ACK="<<s.ack_num()<<",off="<<s._hdr_off<<
    " CTL=["<<(s._urg ? "U" : "") <<(s._ack ? "A" : "") <<
    (s._psh ? "P" : "") <<(s._rst ? "R" : "") <<(s._syn ? "S" : "") <<
    (s._fin ? "F" : "") <<"],wnd="<<s.wnd()<<",chk="<<s.chksum()<<"]\n";
    return db;
    }

    // Pseudo header for checksum calculations
    struct Pseudo_Header {
        u32 src_ip, dst_ip;
        u8 zero, protocol;
        u16 length;

        Pseudo_Header(u32 src,u32 dst,u16 len)
        : src_ip(src), dst_ip(dst), zero(0), protocol(ID_TCP),
          length(CPU::htons(len)) {};
    };
// Compact bitfields istead of using 1 char for each single bit attribute
} __attribute__((packed));

class TCP::Socket_SingleNIC {
public:
    TCP * tcp() const { return TCP::instance(); }
protected:
    Socket_SingleNIC(TCP * tcp) {}
};

class TCP::Socket_MultiNIC {
public:
    TCP * tcp() const { return _tcp; }
protected:
    Socket_MultiNIC(TCP * tcp) : _tcp(tcp) {
        if (!tcp) _tcp = TCP::instance();
    }
private:
    TCP * _tcp;
};

/**
 * The TCP::Socket is a base class for the event driven communication API.
```

```
 * For correct usage the application should use the ClientSocket
 * or ServerSocket instead of this class.
 */
class TCP::Socket : public IF<IP::multiNIC,                                    160
                              Socket_MultiNIC, Socket_SingleNIC>::Result,
                    public Data_Observer<TCP::Address>,
                    public Handler
{
public:
    typedef IF<IP::multiNIC,
               Socket_MultiNIC, Socket_SingleNIC>::Result Base;
    typedef void (Socket::* Handler)(const Header&,const char*,u16);

    enum { // Erros                                                           170
        ERR_NOT_CONNECTED = 1,
        ERR_TIMEOUT,
        ERR_RESET,
        ERR_CLOSING,
        ERR_NO_ROUTE,
        ERR_NOT_STARTED,
        ERR_REFUSED,
        ERR_ILEGAL
    };
                                                                               180
    Socket(const Socket& socket);
    Socket(const Address &local,const Address &remote,TCP * tcp);
    virtual ~Socket();

    // Data_Observer callback
    void update(Data_Observed<TCP::Address> *ob, long c, TCP::Address src,
            TCP::Address dst, void *data, unsigned int size);

    //* Called when connection handshake is complete
    virtual void connected() {};                                               190

    //* Called when data arrives
    virtual void received(const char* data,u16 size) {};

    //* Called when the peer signal a push flag
    virtual void push() {};

    //* Called when data was sucessfuly sent
    virtual void sent(u16 size) {};
                                                                               200
    //* Called when there is an error
    virtual void error(short errorcode) {};

    //* Called when the connection is closed
    // Default action deletes the Socket object
    virtual void closed() { delete this; };

    //* Called when the peer closed his side of the connection
    virtual void closing() {}
                                                                               210
    //* Called to notify an incoming connection
    //* Should return a copy of (or) itself to accept the connection
    virtual Socket* incoming(const Address& from) { return this; }
```

```
    void abort();
    void close();
    void connect();
    void listen();
    void send(const char* data,u16 len,bool push = true);

    const Address & remote() { return _remote; }
    const Address & local() { return _local; }

    enum { // Connection state
        LISTEN,    SYN_SENT,
        SYN_RCVD, ESTABLISHED,
        FIN_WAIT1, FIN_WAIT2,
        CLOSE_WAIT, CLOSING,
        LAST_ACK, TIME_WAIT,
        CLOSED,
    };

    u8 state() { return _state; }
    void state(u8 s) {
        _state = s;
        state_handler = handlers[s];
    }

    // state-processing functions
    void __LISTEN(const Header&,const char*,u16);
    void __SYN_SENT(const Header&,const char*,u16);
    void __SYN_RCVD(const Header&,const char*,u16);
    void __ESTABLISHED(const Header&,const char*,u16);
    void __FIN_WAIT1(const Header&,const char*,u16);
    void __FIN_WAIT2(const Header&,const char*,u16);
    void __CLOSE_WAIT(const Header&,const char*,u16);
    void __CLOSING(const Header&,const char*,u16);
    void __LAST_ACK(const Header&,const char*,u16);
    void __TIME_WAIT(const Header&,const char*,u16);
    void __CLOSED(const Header&,const char*,u16);
    void __SNDING(const Header&,const char*,u16);
    void __RCVING(const Header&,const char*,u16);

    Handler state_handler;

    void operator()();
protected:
    // methods
    bool check_seq(const Header &h,u16 len);
    void send_ack();
    void send_fin();
    void send_reset();
    s32 _send(Header * hdr, SegmentedBuffer * sb);
    void set_timeout();
    void clear_timeout();

    // attributes
    TCP::Address _remote;
    TCP::Address _local;
    volatile u8 _state;
    volatile u32 _rtt;
```

```
    volatile u32 snd_una, snd_nxt, snd_ini, snd_wnd;
    volatile u32 rcv_nxt, rcv_ini, rcv_wnd;

    Alarm * _timeout;
    char _timeout_alloc[sizeof(Alarm)];

    // class attributes
    static Handler handlers[13];                                          280
};

u16 TCP::mss() {
   return ip()->nic()->mtu() - sizeof(TCP::Header) - sizeof(IP::Header);
}

/**
 * The ClientSocket represents an active communicator using the
 * TCP/IP subsystem.
 *                                                                       290
 * Applications using this class must implement all pure virtual methods
 * from TCP::Socket.
 */
class TCP::ClientSocket : public TCP::Socket {
public:

    /**
     * Creates an active socket bound to address _local_ and remote peer
     * address _remote_. With _start_ = true a connection attempt will
     * be made as soon as the object is created.                         300
     */
    ClientSocket(const Address &remote,const Address &local,
                 bool start = true,TCP * tcp = 0);
    virtual ~ClientSocket() {}

    void connect() { Socket::connect(); }
    void connect(const Address& to) {
        _local = to;
        Socket::connect();
    }                                                                     310
};

/**
 * The ServerSocket represents a passive communicator using the
 * TCP/IP subsystem.
 *
 * Applications using this class (by inheritance) must implement
 * all pure virtual methods from TCP::Socket.
 */
class TCP::ServerSocket : public TCP::Socket {                            320
public:

    /**
     * Creates a passive socket that listens on the IP:PORT specified by _local_
     * If _start_ is true, the socket will start listening right after the
     * object creation, otherwise the implementor should call listen().
     */
    ServerSocket(const Address &local,bool start = true,TCP * tcp = 0);
    ServerSocket(const TCP::ServerSocket &socket);
    virtual ~ServerSocket() {}                                            330
```

```
};


/**
 * TCP::Channel is a blocking (synchronous) API for communication using
 * the TCP/IP subsystem. It is built upon the event-driven TCP::Socket
 * to offer a simple send/receive, connect/listen, stream-based, comm.
 * framework.
 *
 * While the TCP::Socket use is done by inheritance, TCP::Channel can be
 * used by composition/agregation.
 */
class TCP::Channel : public ICMP::Observer, public TCP::Socket {
public:
    typedef Alarm::Microsecond Microsecond;

    Channel();
    virtual ~Channel();

    /**
     * Waits for incoming data.
     * return number of bytes received
*/
    int receive(char * dst,unsigned int size);

    /**
     * Sends _size_ bytes from _src_ to the remote peer.
     * Returns upon acknownledge or timeout.
     * return number of bytes sent.
*/
    int send(const char * src,unsigned int size);

    /**
     * Connect to remote host.
     * This method blocks until the connection is established.
     */
    bool connect(const TCP::Address& to);

    /**
     * Associate this channel with a local port.
     */
    void bind(unsigned short port);

    /**
     * Closes the connection.
     * Attention: The meaning of the "CLOSE" function in the
     * BSD Sockets API is: "I will not send more data, but can receive"
     * This method DOES NOT HAVE the same meaning. It will block until
     * the connection is fully closed or aborted.
     */
    bool close();

    /**
     * Wait for a remote peer to connect.
     * return true if connected, false otherwise
*/
    bool listen();
```

```
    /**
     * Get error condition. Zero means no error.
     */
    short error() { return _error; }

protected:
    // clear internal attributes
    void clear();

    // from TCP::Socket

    void received(const char* data,u16 size);
    void closing();
    void closed();
    void connected();
    void sent(u16 size);
    void error(short errorcode);
    void push();

    // ICMP callback
    void update(Data_Observed<IP::Address> *ob, long c,
                IP::Address src, IP::Address dst,
                void *data, unsigned int size);

    // Attributes

    bool _sending;
    bool _receiving;

    Condition _rx_block;
    Condition _tx_block;

    char * _rx_buffer_ptr;
    unsigned int _rx_buffer_size;
    volatile unsigned int _rx_buffer_used;

    volatile unsigned int _tx_bytes_sent;

    volatile short _error;
};

__END_SYS
#endif
```

A.2 Código-fonte

A.2.1 ip.cc

```
#include <ip.h>

__BEGIN_SYS
```

```
IP* IP::_instance[Traits<NIC>::NICS::Length];
const IP::Address IP::NULL = IP::Address((u32)0);

u16 IP::Header::pktid = 0; // incremental packet id

// IP::Address                                                          10

IP_Address::IP_Address(const char * _addr) {
    unsigned char addr[4];
    addr[0] = 0; addr[1] = 0; addr[2] = 0; addr[3] = 0;
    int i;
    for(i=0;i<4;++i) {
        char * sep = strchr(_addr,'.');
        addr[i] = atol(_addr);
        if (!sep) break;
        _addr = ++sep;                                                  20
    }
    memcpy(this,addr,sizeof(this));
}

char* IP_Address::to_string(char * dst) {
    const u8 * _addr = reinterpret_cast<const u8*>(this);
    char* p = dst;
    for(int i=0;i<4;i++) {
        p += utoa(_addr[i], p);
        *p++ = '.';                                                     30
    }
    // remove last dot
    --p;
    *p = 0;
    return p;
}

Debug& operator<<(Debug& db,const IP_Address& addr) {
   const u8 * _addr = reinterpret_cast<const u8*>(&addr);
   db << dec << (int)(_addr[0]) << "." << (int)(_addr[1])               40
      << "." << (int)(_addr[2]) << "." << (int)(_addr[3]);
   return db;
}

// IP::Header

void IP::Header::calculate_checksum() {
    _checksum = 0;
    _checksum = ~(IP::calculate_checksum(this, hlength()));
}                                                                       50

// IP
IP::IP(unsigned int unit)
    : _nic(unit),
      _network_service(&_nic, this),
      _self(IP::NULL),
      _broadcast(255,255,255,255),
      _thread(0)
{
    if (_instance[unit])                                                60
    {
        db<IP>(ERR) << "IP::created IP object twice for the same NIC!";
```

```
    }
    _network_service.update(_broadcast, NIC::BROADCAST);

    if (CONFIG == STATIC) {
        _self = Address(ADDRESS);
        _network_service.update(_self,_nic.address());
        _broadcast = Address(BROADCAST);
        _network_service.update(_broadcast, NIC::BROADCAST);
        _netmask = Address(NETMASK);
    }

    _instance[unit] = this;

    // allocate memory for receiving packets
    for(unsigned int i=0;i<MAX_FRAGMENTS;++i)
        _packet_receive[i] = new (kmalloc(mtu())) char[mtu()];

    _packet_send = new (kmalloc(mtu())) char[mtu()];

    start();
}

IP::~IP() {
    for(unsigned int i=0;i<MAX_FRAGMENTS;++i)
        kfree(_packet_receive[i]);

    kfree(_packet_send);
}

void IP::process_ip(char *data, u16 size)
{
    Header &pck_h = *reinterpret_cast<Header*>(data);
    if((u32)_self != (u32)0 && // We MUST accept anything if our IP address is not set
       (u32)(pck_h.dst_ip()) != (u32)(_self) &&
       (u32)(pck_h.dst_ip()) != (u32)(_broadcast))
    {
        db<IP>(INF) << "IP Packet discarded. dst= " << pck_h.dst_ip() << "\n";
        return;
    }
    else {
        db<IP>(TRC) << "IP: " << pck_h << "\n" ;
    }

    if(!fragmentation && (pck_h.flags() == Header::MF_FLAG || pck_h.offset() != 0))
    {
        db<IP>(INF) << "IP::Fragmented packet discarded\n";
        return;
    }

    if (calculate_checksum(data,pck_h.hlength()) != 0xFFFF) {
        db<IP>(TRC) << "IP checksum failed for incoming packet\n";
    } else {
        notify(pck_h.src_ip(),pck_h.dst_ip(),(int)pck_h.protocol(),
                &data[pck_h.hlength()], pck_h.length() - pck_h.hlength());
        if (pck_h.ttl() > 0) {
            pck_h.ttl(pck_h.ttl() - 1);
        }
    }
```

```
}

int IP::run()
{
    db<IP>(TRC) << __PRETTY_FUNCTION__ << endl;
    NIC::Address src;
    NIC::Protocol prot;

    while (true) {                                                          130
        int size = _nic.receive(&src, &prot, _packet_receive[0], _nic.mtu());

        if(size <= 0) {
            //db<IP>(WRN) << "NIC::received error!" << endl;
            Thread::self()->yield();
            continue;
        }

        if (prot == NIC::IP) {
            _network_service.update(reinterpret_cast<Header*>(_packet_receive[0])->src_ip(), src);   140
            process_ip(_packet_receive[0], size);
        }

        // notify routing algorithm
        _network_service.received(src, prot, _packet_receive[0], size);

        Thread::yield();
    }

    return 0;                                                               150
}

s32 IP::send(const Address & from,const Address & to,SegmentedBuffer * data,Protocol proto)
{
    Header hdr(from,to,proto,data->total_size());
    SegmentedBuffer pdu(&hdr,hdr.hlength(),data);

    MAC_Address mac = NIC::BROADCAST;
    if (from.is_neighbor(to,_netmask))
        mac = _network_service.resolve(to,&pdu);                            160
    else
        mac = _network_service.resolve(_gateway,&pdu);

    //TODO: put fragmentation here
    int size = pdu.total_size();
    pdu.copy_to(_packet_send,size);

    db<IP>(TRC) << "IP::send() " << size << " bytes" << endl;

    int retry = 5, ret;                                                     170

    do {
        ret = _nic.send(mac,NIC::IP,_packet_send,size);
        if (ret >= 0)
            return size;

        Thread::self()->yield();
    } while (retry-- > 0);
```

```
    return -1;
}

// From http://www.faqs.org/rfcs/rfc1071.html
u16 IP::calculate_checksum(const void* ptr, u16 count)
{
    u32 sum = 0;

    const unsigned char * _ptr = reinterpret_cast<const unsigned char *>(ptr);
    u16 i;

    for(i = 0; i < count-1; i+=2)
        sum += (((unsigned short)(_ptr[i+1]) & 0x00FF) << 8) | _ptr[i];
    if(count & 1) {
        sum += _ptr[count-1];
    }

    while(sum >> 16)
        sum = (sum & 0xffff) + (sum >> 16);

    return sum;
}

__END_SYS
```

A.2.2 icmp.cc

```
#include <icmp.h>

__BEGIN_SYS

ICMP * ICMP_SingleNIC::_instance;

ICMP * ICMP_MultiNIC::_instance[Traits<NIC>::NICS::Length];

ICMP * ICMP_SingleNIC::instance()
{
    if (!_instance)
        _instance = new ICMP(IP::instance());
    return _instance;
}

ICMP * ICMP_MultiNIC::instance(unsigned int i)
{
    if (!_instance[i])
        _instance[i] = new ICMP(IP::instance(i));
    return _instance[i];
}

ICMP::ICMP(IP* _ip) : Base(_ip)
{
    ip()->attach(this, ICMP_ID);
}
```

```
ICMP::~ICMP()
{
    ip()->detach(this, ICMP_ID);
}

void ICMP::update(Data_Observed<IP::Address> *ob, long c, IP::Address src,
                  IP::Address dst, void *data, unsigned int size)
{
    Packet& packet = *reinterpret_cast<Packet*>(data);
    if (IP::calculate_checksum(data,size) != 0xFFFF) {
        db<ICMP>(TRC) << "ICMP::checksum error\n";
        return;
    }

    if (Traits<ICMP>::echo_reply && (packet.type() == ECHO)) { // PONG
        db<ICMP>(TRC) << "ICMP::echo sending automatic reply to " << src << endl;
        Packet reply(ECHO_REPLY,0,packet.id(),packet.sequence(),packet._data);
        send(dst,src,reply);
    }

    if (packet.type() == ECHO_REPLY) {
        db<ICMP>(TRC) << "ICMP::echo reply from " << src << endl;
    }

    notify(src,dst,packet.type(),data,size);
}

ICMP::Packet::Packet(Type type,Code code, unsigned short id,unsigned short seq,
                     const char * data,short size)
    : _type(type),
      _code(code),
      _checksum(0),
      _id(CPU::htons(id)),
      _sequence(CPU::htons(seq))
{
    if (data) memcpy(_data,data,size < 56 ? size : 56);
    else memset(_data, 0, 56);
}

void ICMP::send(IP::Address from,IP::Address to,Packet& pkt)
{
    // Thou shall not calculate the checksum in ctor body!
    pkt._checksum = 0;
    pkt._checksum = ~(IP::calculate_checksum(&pkt, sizeof(pkt)));
    SegmentedBuffer sb(pkt.raw(),sizeof(pkt));
    ip()->send(from,to,&sb,ICMP_ID);
}

__END_SYS
```

A.2.3 dhcp.cc

```
#include <dhcp.h>
```

```
__BEGIN_SYS

DHCP::Client::Client(UDP * udp)
    : UDP::Socket(UDP::Address(0,68),UDP::Address(~0,67), udp),
      _state(IDLE)
    { }

                                                                    10

void DHCP::Client::configure() {
    db<IP>(INF) << "DHCP::Client sending discover msg\n";
    _state = DISCOVER;

    DHCP::Discover pkt(udp()->ip());
    SegmentedBuffer sb(&pkt,sizeof(DHCP::Discover));

    _xid = pkt.xid();
    send(&sb);                                                      20
}

void DHCP::Client::parse_options(const Packet<255> * packet) {
    db<IP>(TRC) << "DHCP::Parsing OPTIONS\n";

    u8 * opt = packet->options();
    int i;

    for(i=0;i < 255;i++) {
        switch(opt[i]) {                                            30
        case 0: // padding
            break;
        case 1: // netmask
            ++i;
            if (opt[i] == 4) // IPv4, good
            {
                _mask = IP::Address(opt[i+1],opt[i+2],opt[i+3],opt[i+4]);
                db<IP>(TRC) << "Found netmask " << _mask << endl;
            }
            i += opt[i];                                            40
            break;
        case 3: // routers
            ++i;
            if (opt[i] >= 4) // one or more, let's get the first
            {
                _gw = IP::Address(opt[i+1],opt[i+2],opt[i+3],opt[i+4]);
                db<IP>(TRC) << "Found gateway " << _gw << endl;
            }
            i += opt[i];
            break;                                                  50
        case 6: // nameserver
            ++i;
            if (opt[i] >= 4) // same logic as routers
            {
                _ns = IP::Address(opt[i+1],opt[i+2],opt[i+3],opt[i+4]);
                db<IP>(TRC) << "Found nameserver " << _ns << endl;
            }
            i += opt[i];
            break;
        case 28: // broadcast address                               60
```

```
            ++i;
            if (opt[i] == 4) // IPv4, good
            {
                _bcast = IP::Address(opt[i+1],opt[i+2],opt[i+3],opt[i+4]);
                db<IP>(TRC) << "Found bcast " << _bcast << endl;
            }
            i += opt[i];
            break;
        case 51: // lease time in secs
            ++i;                                                                    70
            if (opt[i] == 4) { // Good size!
                _lease_time = (((u32)(opt[i+1]) << 24) & 0xFF000000) |
                              (((u32)(opt[i+2]) << 16) & 0x00FF0000) |
                              (((u32)(opt[i+3]) << 8 ) & 0x0000FF00) |
                              (((u32)(opt[i+4])      ) & 0x000000FF);
                db<IP>(TRC) << "Lease time " << _lease_time << endl;
            }
            i += opt[i];
            break;
        case 255: // end                                                           80
            i = 500; // get out of the loop
            {
                db<IP>(TRC) << "End of options " << endl;
            }
            break;
        default:
            {
            db<IP>(TRC) << "Skipping code " << (int)opt[i] << " len: " << opt[i+1] << endl;
                i += opt[i+1] + 1;
            }                                                                       90

        }
    }
}

void DHCP::Client::received(const UDP::Address & src,
                            const char *data, unsigned int size)
{
    db<IP>(INF) << "DHCP::Client state: " << _state << "\n";
    const DHCP::Packet<255> * packet = reinterpret_cast<const DHCP::Packet<255> *>(data);   100

    if (_xid != packet->xid()) {
        db<IP>(TRC) << "This DHCP message does not belong to me\n";
    }

    switch(_state) {
    case DISCOVER:

        if (packet->your_address()) {
            remote(src);                                                            110

            UDP::Address me(packet->your_address(),68);
        //  set_local(me);
        //  _udp->ip()->set_address(me.ip());

            _state = REQUEST;
            db<IP>(INF) << "Server " << src.ip() << " offered IP " << me.ip() << "\n";
            parse_options(packet);
```

```
            DHCP::Request req(udp()->ip(),packet);
            SegmentedBuffer sb(&req,sizeof(DHCP::Request));
            send(&sb);
        }

        break;

    case REQUEST:

        if (packet->your_address()) {
            _ip = IP::Address((u32)packet->your_address());

            parse_options(packet);
            _state = RENEW;

        }
    }
}

__END_SYS
```

A.2.4 udp.cc

```
#include <udp.h>

#include <utility/string.h> // for Address constructor

__BEGIN_SYS

// UDP::Address
UDP_Address::UDP_Address(const char *addr) : _ip(addr)
{
    char *sep = strchr(addr,':');
    if (sep) {
        _port = atol(++sep);
    } else {
        _port = 0;
    }
}

char* UDP_Address::to_string(char * dst)
{
    char *p = _ip.to_string(dst);
    *p++ = ':';
    p += utoa(_port,p);
    *p = 0;
    return p;
}


// TCP's checksum was added to UDP, we should merge this to avoid code redundancy
void UDP::Header::checksum(IP::Address src,IP::Address dst,SegmentedBuffer * sb)
{
    if (!Traits<UDP>::checksum) {
```

```
        _checksum = 0;
        return;
    }

    db<UDP>(TRC) << __PRETTY_FUNCTION__ << endl;
    u16 len;
    len = sizeof(this);

    if (sb) len += sb->total_size();

    Pseudo_Header phdr((u32)src,(u32)dst,len);

    _checksum = 0;

    u32 sum = 0;

    sum = IP::calculate_checksum(&phdr, sizeof(phdr));
    sum += IP::calculate_checksum(this, sizeof(this));

    while (sb) {
        sum += IP::calculate_checksum(sb->data(), sb->size());
        sb = sb->next();
    }

    while (sum >> 16)
        sum = (sum & 0xFFFF) + (sum >> 16);

    _checksum = ~sum;
}

UDP::UDP(IP * _ip) : Base(_ip) {
    ip()->attach(this, ID_UDP);
}

UDP::~UDP() {
    ip()->detach(this, ID_UDP);
}

UDP::Socket::Socket(Address local, Address remote, UDP * _udp)
    : Base(_udp), _local(local), _remote(remote)
{
    udp()->attach(this, _local.port());
}

UDP::Socket::~Socket() {
    udp()->detach(this, _local.port());
}

// Assembles data and sends to IP layer

s32 UDP::send(Address _local, Address _remote, SegmentedBuffer * data) {
    UDP::Header hdr(_local.port(), _remote.port(),
                    data->total_size());
    SegmentedBuffer sb(&hdr, sizeof(UDP::Header), data);
    hdr.checksum(_local.ip(),_remote.ip(),&sb);
    return ip()->send(_local.ip(), _remote.ip(), &sb, ID_UDP) - 8;    // discard header
}
```

```
// Called by IP's notify(...)

void UDP::update(Data_Observed<IP::Address> *ob, long c, IP::Address src,
                 IP::Address dst, void *data, unsigned int size)
{
    Header& hdr = *reinterpret_cast<Header*>(data);

    db<UDP>(INF) << "UDP::update: received "<< size <<" bytes from "
                 << src << " to " << dst << "\n";

    if (Traits<UDP>::checksum && hdr._checksum != 0) {
        SegmentedBuffer sb(static_cast<char*>(data) + sizeof(Header), size - sizeof(Header));
        u16 csum = hdr._checksum;
        hdr.checksum(src,dst,&sb);
        if (hdr._checksum != csum) {
            db<UDP>(INF) << "UDP::checksum failed for incomming data\n";
            return;
        }
    }
    notify(UDP::Address(src,hdr.src_port()),UDP::Address(dst,hdr.dst_port()),
           (int) hdr.dst_port(), &((char*)data)[sizeof(Header)],
           size - sizeof(Header));
}

UDP * UDP::instance(unsigned int i) {
    static UDP * _instance[Traits<NIC>::NICS::Length];
    if (!_instance[i])
        _instance[i] = new UDP(IP::instance(i));
    return _instance[i];
}

// Called by UDP's notify(...)

void UDP::Socket::update(Observed *o, long c, UDP_Address src, UDP_Address dst,
                         void *data, unsigned int size)
{
    // virtual call
    received(src,(const char*)data,size);
}

// UDP Channel

int UDP::Channel::receive(Address * from,char * buf,unsigned int size)
{
    _buffer_size = size;
    _buffer_data = buf;
    _buffer_src  = from;
    _buffer_wait.wait();
    _buffer_data = 0;
    return _buffer_size;
}

void UDP::Channel::received(const Address & src,
                            const char *data, unsigned int size)
{
    if (_buffer_data) {
        if (size < _buffer_size)
            _buffer_size = size;
```

```
        memcpy(_buffer_data, data, _buffer_size);
        memcpy(_buffer_src, &src, sizeof(Address));
        _buffer_wait.signal();
    }
}

void UDP::Channel::update(Data_Observed<IP::Address> *ob, long c,
                          IP::Address src, IP::Address dst,
                          void *data, unsigned int size)
{
    ICMP::Packet& packet = *reinterpret_cast<ICMP::Packet*>(data);
    if (packet.type() == ICMP::UNREACHABLE)
    {
        IP::Header& ip_hdr = *reinterpret_cast<IP::Header*>(packet.data());
        if (ip_hdr.src_ip() != (u32)_local.ip() ||
            ip_hdr.dst_ip() != (u32)_remote.ip()) {
            return;
        }
        char * ip_data = (char*)data + ip_hdr.hlength();
        UDP::Header& udp_hdr = *reinterpret_cast<UDP::Header*>(ip_data);
        if (udp_hdr.src_port() != _local.port() ||
            udp_hdr.dst_port() != _remote.port()) {
            return;
        }

        _error = ~(packet.code());
    }
}

UDP::Channel_MultiNIC::Channel_MultiNIC()
{
    int i;
    /*
     * Attention here:
     * The correct is static_cast<>(), but it doesn't work
     * if we are in a single-NIC case. For this reason C-style cast
     * is used since it falls back to reinterpret_cast and this
     * code will not be used in single-NIC scenario anyway.
     */
    IP * _ip = ((Channel*)this)->udp()->ip();
    for(i=0;i < Traits<NIC>::NICS::Length; ++i)
    {
        if (IP::instance(i) == _ip) {
            _icmp = ICMP_MultiNIC::instance(i);
            break;
        }
    }
}

void UDP::Channel::operator()()
{
    _buffer_data = 0;
    _buffer_size = TIMEOUT;
    _buffer_wait.signal();
}

UDP::Channel::Channel(const Address& local,const Address& remote)
//TODO: mult NIC support for channels
```

```
    : Socket(local, remote, 0), _error(0)
{
    icmp()->attach(this, ICMP::UNREACHABLE);
}

UDP::Channel::~Channel()
{
    if (_buffer_data) {
        db<UDP>(ERR) << "UDP::Channel for "<<this<<" destroyed while receiving\n";
        _buffer_size = DESTROYED;
        _buffer_wait.signal();
    }
    icmp()->detach(this, ICMP::UNREACHABLE);
}

__END_SYS
```

A.2.5 tcp.cc

```
#include <tcp.h>

__BEGIN_SYS

// static data
TCP::Socket::Handler TCP::Socket::handlers[13] = {
    &TCP::Socket::__LISTEN,   &TCP::Socket::__SYN_SENT,
    &TCP::Socket::__SYN_RCVD, &TCP::Socket::__ESTABLISHED,
    &TCP::Socket::__FIN_WAIT1, &TCP::Socket::__FIN_WAIT2,
    &TCP::Socket::__CLOSE_WAIT, &TCP::Socket::__CLOSING,
    &TCP::Socket::__LAST_ACK, &TCP::Socket::__TIME_WAIT,
    &TCP::Socket::__CLOSED };

TCP::TCP(IP * _ip) : Base(_ip)
{
    ip()->attach(this, ID_TCP);
}

TCP::~TCP()
{
    ip()->detach(this, ID_TCP);
}

TCP * TCP::instance(unsigned int i) {
    static TCP * _instance[Traits<NIC>::NICS::Length];
    if (!_instance[i])
        _instance[i] = new TCP(IP::instance(i));
    return _instance[i];
}

// Called by IP's notify(...)

void TCP::update(Data_Observed<IP::Address> *ob, long c, IP::Address src,
                 IP::Address dst, void *data, unsigned int size)
{
```

```
    Header& hdr = *reinterpret_cast<Header*>(data);

    db<TCP>(TRC) << "TCP::update: "<< hdr << endl;

    if (!(hdr.validate_checksum(src,dst,size - hdr.size()))) {
        db<TCP>(INF) << "TCP checksum failed for incoming packet!\n";
        return;
    }

    int len = size - hdr.size();
    if (len < 0) {
        db<TCP>(INF) << "Misformed TCP segment received\n";
        return;
    }

    notify(TCP::Address(src,hdr.src_port()),
           TCP::Address(dst,hdr.dst_port()),
           (int) hdr.dst_port(), data, size);
}


// Called by TCP's notify(...)

void TCP::Socket::update(Data_Observed<TCP::Address> *o, long c, TCP::Address src,
                         TCP::Address dst, void *data, unsigned int size)
{
    Header& hdr = *reinterpret_cast<Header*>(data);
    int len = size - hdr.size();

    if ((_remote == src) || (_remote.port() == 0))
    {
        if (state() == LISTEN) _remote = src;
        (this->*state_handler)(hdr,&((char*)data)[hdr.size()],len);
    } else {
        db<TCP>(TRC) << "TCP Segment does not belong to us\n";
    }
}

// Header stuff

TCP::Header::Header(u32 seq,u32 ack)
{
    memset(this,0,sizeof(Header));
    seq_num(seq);
    ack_num(ack);
}

bool TCP::Header::validate_checksum(IP::Address src,IP::Address dst,u16 len)
{
    len += size();

    Pseudo_Header phdr((u32)src,(u32)dst,len);

    u32 sum = 0;

    sum = IP::calculate_checksum(this, len);
    sum += IP::calculate_checksum(&phdr, sizeof(phdr));
```

```
    while (sum >> 16)
        sum = (sum & 0xFFFF) + (sum >> 16);

    return sum == 0xFFFF;
}

void TCP::Header::_checksum(IP::Address src,IP::Address dst,SegmentedBuffer * sb)
{
    u16 len;
    len = size();

    if (sb) len += sb->total_size();

    Pseudo_Header phdr((u32)src,(u32)dst,len);

    _chksum = 0;

    u32 sum = 0;

    sum = IP::calculate_checksum(&phdr, sizeof(phdr));
    sum += IP::calculate_checksum(this, size());

    while (sb) {
        sum += IP::calculate_checksum(sb->data(), sb->size());
        sb = sb->next();
    }

    while (sum >> 16)
        sum = (sum & 0xFFFF) + (sum >> 16);

    _chksum = ~sum;
}

// Socket stuff

TCP::Socket::Socket(const Address &remote,const Address &local,TCP * _tcp)
    : Base(_tcp), _remote(remote), _local(local), _rtt(500000), _timeout(0)
{
    rcv_wnd = tcp()->mss();
    state(CLOSED);
    tcp()->attach(this, _local.port());
}

TCP::Socket::Socket(const TCP::Socket& socket) : Base(socket.tcp())
{
    memcpy(this, &socket, sizeof(Socket));
    state(CLOSED);
    tcp()->attach(this, _local.port());
}

TCP::Socket::~Socket()
{
    tcp()->detach(this, _local.port());
    clear_timeout();
}

s32 TCP::Socket::_send(Header * hdr, SegmentedBuffer * sb)
{
```

```
    // fill header
    hdr->src_port(_local.port());
    hdr->dst_port(_remote.port());
    hdr->_hdr_off = 5; // our header is always 20 bytes
    hdr->wnd(rcv_wnd);
    hdr->chksum(0);
    hdr->_checksum(_local.ip(),_remote.ip(),sb);

    // hdr + sb                                                                    160
    SegmentedBuffer nsb(hdr,hdr->size());
    nsb.append(sb);

    return tcp()->ip()->send(_local.ip(),_remote.ip(),&nsb,TCP::ID_TCP) - hdr->size();
}

/**
 * The basic Socket::send() splits data into multiple segments
 * respecting snd_wnd and mtu.
 */                                                                              170
void TCP::Socket::send(const char *data,u16 len,bool push)
{
    if (snd_wnd == 0) { // peer cannot receive data
        set_timeout();
        send_ack(); // zero-window probing
        return;
    }
    // cannot send more than the peer is willing to receive
    if (len > snd_wnd)
        len = snd_wnd;                                                           180

    int more = 0,mss = tcp()->mss();
    if (len > mss) { // break up into multiple segments
        more = len - mss;
        len = mss;
    }

    Header hdr(snd_nxt,rcv_nxt);
    hdr._ack = true; // with pidgeback ack
    hdr._psh = (push && (more == 0)); // set push flag only on the last segment   190
    snd_nxt += len;
    SegmentedBuffer sb(data,len);

    _send(&hdr,&sb);

    //if (more > 0) // send next segment too //TODO: TCP::Channel also do it!
    //    send(&data[len], more);

    set_timeout();
}                                                                                200

void TCP::Socket::set_timeout() {
    // the needed logic to finish an alarm is int the destructor
    // but we use preallocated memory, so we cannot use 'delete'
    if (_timeout)
        _timeout->Alarm::~Alarm();
    _timeout = new (&_timeout_alloc) Alarm(2 * _rtt, this, 1);
}
```

```
void TCP::Socket::clear_timeout() {
    if (_timeout)
    {
        _timeout->Alarm::~Alarm();
        _timeout = 0;
    }
}

void TCP::Socket::operator()() {
    _timeout->Alarm::~Alarm();
    _timeout = 0;

    snd_nxt = snd_una; // rollback, so the user can resend
    error(ERR_TIMEOUT);
}

void TCP::Socket::close()
{
    send_fin();
    set_timeout();
    if (state() == ESTABLISHED)
        state(FIN_WAIT1);
    else if (state() == CLOSE_WAIT) {
        state(LAST_ACK);
    }
    else if (state() == SYN_SENT) {
        state(CLOSED);
        clear_timeout();
        closed();
    }
}


TCP::ClientSocket::ClientSocket(const Address& remote,const Address& local,
                                bool start, TCP * tcp)
        : Socket(remote,local,tcp)
{
    if (start)
        Socket::connect();
}

void TCP::Socket::connect() {
    if (state() != CLOSED && state() != SYN_SENT) {
        db<TCP>(ERR) << "TCP::Socket::connect() could not be called\n";
        return;
    }

    state(SYN_SENT);
    snd_ini = Pseudo_Random::random() & 0x00FFFFFF;
    snd_una = snd_ini;
    snd_nxt = snd_ini + 1;

    Header hdr(snd_ini, 0);
    hdr._syn = true;
    _send(&hdr,0);
    set_timeout();
}
```

```
TCP::ServerSocket::ServerSocket(const Address& local,bool start,TCP * tcp)
    : Socket(Address(0,0),local,tcp)
{                                                                          270
    if (start)
        Socket::listen();
}

TCP::ServerSocket::ServerSocket(const TCP::ServerSocket &socket)
    : Socket(socket)
{
}

void TCP::Socket::listen()                                                 280
{
    if (state() != CLOSED && state() != LISTEN) {
        db<TCP>(ERR) << "TCP::Socket::listen() called with state: " << state() << endl;
        return;
    }

    _remote = Address(0,0);
    state(LISTEN);
}
                                                                           290
void TCP::Socket::__LISTEN(const Header& r ,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;

    if (r._syn && !r._rst && !r._fin) {
        Socket * n;

        if ((n = incoming(_remote)) != 0) {
            n->_remote = _remote;
            n->rcv_nxt = r.seq_num()+1;                                    300
            n->rcv_ini = r.seq_num();
            n->snd_wnd = r.wnd();

            n->state(SYN_RCVD);

            n->snd_ini = Pseudo_Random::random() & 0x0000FFFF;

            Header s(n->snd_ini,n->rcv_nxt);
            s._syn = true;
            s._ack = true;                                                 310
            n->_send(&s,0);

            n->snd_nxt = n->snd_ini+1;
            n->snd_una = n->snd_ini;

            n->set_timeout();
        }
        // else = connection rejected
    }
    if (state() == LISTEN) {                                               320
        // a new socket was created to handle the incomming connection
        // and we stay in the listening state
        _remote = Address((u32)0,(u16)0);
    }
}
```

```
void TCP::Socket::__SYN_SENT(const Header& r,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;

    if (r._rst || r._fin) {
            error(ERR_REFUSED);
            state(CLOSED);
            clear_timeout();
            closed();
    }
    else if (r._ack) {
        if ((r.ack_num() <= snd_ini) || (r.ack_num() > snd_nxt)) {
            error(ERR_RESET);
            state(CLOSED);
            clear_timeout();
            closed();
        } else if ((r.ack_num() >= snd_una) && (r.ack_num() <= snd_nxt)) {
            if (r._syn) {
                rcv_nxt = r.seq_num() + 1;
                rcv_ini = r.seq_num();
                snd_una = r.ack_num();
                snd_wnd = r.wnd();
                if (snd_una <= snd_ini) {
                    state(SYN_RCVD);
                } else {
                    state(ESTABLISHED);
                    clear_timeout();
                    send_ack();
                    connected();
                }
            } else {
                // TODO: discover what to do here
            }
        }
    } else if (!r._rst && r._syn) {
        rcv_nxt = r.seq_num() + 1;
        snd_ini = r.seq_num();
        snd_wnd = r.wnd();
        state(SYN_RCVD);
    }

}

void TCP::Socket::__SYN_RCVD(const Header& r ,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;

    if (!check_seq(r,len))
        return;

    if (r._rst || r._fin) {
        error(ERR_RESET);
        state(CLOSED);
        clear_timeout();
        closed();
    }
```

```
    else if (r._ack) {
        snd_wnd = r.wnd();
        snd_una = r.ack_num();
        state(ESTABLISHED);
        clear_timeout();
        connected();
    }
}

void TCP::Socket::__RCVING(const Header &r,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;
    if (len) {
        rcv_nxt += len;
        send_ack();
        received(data,len);
        if (r._psh)
            push();
    } else {
        send_ack();
    }
}

void TCP::Socket::__SNDING(const Header &r,const char* data, u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;

    if (r._ack) {
        int bytes = r.ack_num() - snd_una;
        if (bytes < 0) // sliding window overflow
            bytes = r.ack_num() + (0xFFFF - snd_una);
        sent(bytes);
        snd_una = r.ack_num();
        if (snd_una == snd_nxt)
            clear_timeout();
    }
}

void TCP::Socket::__ESTABLISHED(const Header& r ,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;

    if (!check_seq(r,len))
    {
        if ((len) && (r.seq_num() < rcv_nxt))
            send_ack();
        return;
    }

    if (r._rst) {
        error(ERR_RESET);
        state(CLOSED);
        clear_timeout();
        closed();
    }
    else if (r.seq_num() == rcv_nxt) { // implicit reject out-of-order segments
        snd_wnd = r.wnd();
```

```
        if (snd_una < r.ack_num())
            __SNDING(r,data,len);

        if (len)
            __RCVING(r,data,len);

        if (r._fin) {
            send_ack();
            state(CLOSE_WAIT);
            closing();
        }
    }
    else {
        db<TCP>(TRC) << "TCP::out of order segment received\n";
    }
}


void TCP::Socket::__FIN_WAIT1(const Header& r ,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;

    if (!check_seq(r,len))
        return;

    if (!r._fin && len) {
        __RCVING(r,data,len);
        if (r._ack)
            state(FIN_WAIT2);
        return;
    }
    if (r._ack && !r._fin) { // TODO: check snd_una
        rcv_nxt = r.seq_num() + len;
        state(FIN_WAIT2);
        send_ack();
    }
    if (r._ack && r._fin) {
        state(CLOSED); // no TIME_WAIT
        send_ack();
        clear_timeout();
        closed();
    }
    if (!r._ack && r._fin) {
        state(CLOSING);
        send_ack();
    }
}

void TCP::Socket::__FIN_WAIT2(const Header& r ,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;
    if (!check_seq(r,len))
        return;
    if (len) {
        __RCVING(r,data,len);
    }

    if (r._fin) {
```

```
        state(CLOSED); // no TIME_WAIT
        send_ack();
        clear_timeout();
        closed();
    }
}

void TCP::Socket::__CLOSE_WAIT(const Header& r ,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;
    if (!check_seq(r,len))
        return;

    if (r._rst || len) {
        if (len)
            send_reset();
        error(ERR_RESET);
        state(CLOSED);
        clear_timeout();
        closed();
    }
}

void TCP::Socket::__CLOSING(const Header& r ,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;
    if (!check_seq(r,len))
        return;

    if (r._ack) {
        state(CLOSED); // no TIME_WAIT
        clear_timeout();
        closed();
    }
}

void TCP::Socket::__LAST_ACK(const Header& r ,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;
    if (!check_seq(r,len))
        return;
    if (r._ack) {
        state(CLOSED);
        clear_timeout();
        closed();
    }
}

void TCP::Socket::__TIME_WAIT(const Header& r ,const char* data,u16 len)
{
    db<TCP>(TRC) << __PRETTY_FUNCTION__ << endl;
    if (!check_seq(r,len))
        return;

    if (r._fin && r._ack) {
        state(CLOSED);
        clear_timeout();
        closed();
```

```
    }
}

void TCP::Socket::__CLOSED(const Header&,const char*,u16)
{
    // does nothing
}

void TCP::Socket::send_ack()
{
    Header s(snd_nxt,rcv_nxt);
    s._ack = true;
    _send(&s,0);
}

void TCP::Socket::send_fin()
{
    Header s(snd_nxt,rcv_nxt);
    s._fin = true;
    s._ack = true;
    _send(&s,0);
}

void TCP::Socket::send_reset()
{
    Header s(snd_nxt,rcv_nxt);
    s._fin = true;
    s._ack = true;
    s._psh = true;
    s._rst = true;
    _send(&s,0);
}

bool TCP::Socket::check_seq(const Header &h,u16 len)
{
    if ((len <= rcv_wnd) &&
        (h.seq_num() == rcv_nxt))
    {
        return true;
    }

    db<TCP>(TRC) << "TCP: check_seq() == false\n";
    return false;
}

void TCP::Socket::abort()
{
    send_reset();
    clear_timeout();
    state(CLOSED);
}

// Channel stuff

TCP::Channel::Channel()
    : TCP::Socket(TCP::Address(0,0),TCP::Address(0,0),0)
{
    clear();
```

```
    ICMP::instance()->attach(this, ICMP::UNREACHABLE);
}

TCP::Channel::~Channel() {
    if (state() != CLOSED) {
        db<TCP>(ERR) << "Destroying non-closed channel!\n";
        // This condition must REALLY not happen.
    }
    ICMP::instance()->detach(this, ICMP::UNREACHABLE);
}

bool TCP::Channel::connect(const TCP::Address& to)
{
    if (state() != CLOSED) {
        db<TCP>(ERR) << "TCP::Channel::connect() called for open connection!\n";
        return false;
    }

    int retry = 5;
    _remote = to;
    clear();
    _sending = true;
    do {
        Socket::connect();
        _tx_block.wait();
    } while (retry-- > 0 && state() != ESTABLISHED);
    _sending = false;

    if (state() != ESTABLISHED)
    {
        clear_timeout();
        state(CLOSED);
    }

    return state() == ESTABLISHED;
}

int TCP::Channel::receive(char * dst,unsigned int size)
{
    if (_error)
        return -_error;

    if (state() != ESTABLISHED)
        return -ERR_NOT_CONNECTED;

    if (_receiving) {
        db<TCP>(ERR) << "TCP::Channel::receive already called!\n";
        return -ERR_ILEGAL;
    }

    _rx_buffer_ptr  = dst;
    _rx_buffer_size = size;
    _rx_buffer_used = 0;
    _receiving = true;
    rcv_wnd = size;
    send_ack(); // send a window update

    _rx_block.wait();
```

```
    int rcvd = _rx_buffer_used;

    _rx_buffer_ptr  = 0;
    _rx_buffer_size = 0;
    _rx_buffer_used = 0;
    _receiving = false;                                                    680

    if (_error)
        return -_error;

    return rcvd;
}

int TCP::Channel::send(const char * src,unsigned int size)
{
    if (_error)                                                            690
        return -_error;

    if (state() != ESTABLISHED)
        return -ERR_NOT_CONNECTED;

    // congestion control not yet done

    _tx_bytes_sent = 0;
    _sending = true;
                                                                           700
    int offset;

    do {
        offset = _tx_bytes_sent;
        Socket::send(&src[offset], size - offset);

        _tx_block.wait();

        if (state() != ESTABLISHED || _error)
            break;                                                         710
    } while (_tx_bytes_sent < size);

    _sending = false;
    return _tx_bytes_sent;
}

void TCP::Channel::bind(unsigned short port)
{
    if (state() != CLOSED) {
        db<TCP>(ERR) << "Cannot use TCP::Channel::bind() on open connection\n";  720
        return;
    }

    tcp()->detach(this, _local.port());

    _local = TCP::Address(tcp()->ip()->address(), port);

    tcp()->attach(this, port);
}
                                                                           730
bool TCP::Channel::close()
```

```
{
    if (state() == CLOSED)
        return true;

    if (state() == SYN_SENT) {
        _tx_block.signal();
        abort();
        return true;
    }

    if (_receiving) {
        _error = ERR_CLOSING;
        _rx_block.signal();
    }

    int retry = 5;
    _sending = true;

    do {
        Socket::close();
        _tx_block.wait();
    } while (retry-- > 0 && state() != CLOSED);

    _sending = false;

    return state() == CLOSED;
}

bool TCP::Channel::listen()
{
    if (state() != CLOSED) {
        db<TCP>(ERR) << "TCP::Channel::listen() called on non-closed channel\n";
        return false;
    }

    clear();
    _sending = true;

    Socket::listen();
    _tx_block.wait();

    _sending = false;

    if (state() != ESTABLISHED)
    {
        clear_timeout();
        state(CLOSED);
    }

    return state() == ESTABLISHED;
}

void TCP::Channel::clear()
{
    _sending   = false;
    _receiving = false;;
    _rx_buffer_ptr  = 0;
    _rx_buffer_size = 0;
```

```
    _rx_buffer_used = 0;
    _tx_bytes_sent = 0;
    _error = 0;
    rcv_wnd = 0;
}

// Channel's implementation of Socket callbacks

void TCP::Channel::received(const char* data,u16 size)
{
    int remaining = _rx_buffer_size - _rx_buffer_used;

    if (!_rx_buffer_ptr || (remaining == 0)) {
        db<TCP>(WRN) << "Channel::received droping data, no buffer space\n";
        return;
    }
    if (remaining < static_cast<int>(size)) {
        db<TCP>(WRN) << "Channel::received data truncated\n";
        size = static_cast<u16>(remaining);
    }

    memcpy(&_rx_buffer_ptr[_rx_buffer_used], data, size);

    _rx_buffer_used += size;
    rcv_wnd = _rx_buffer_size - _rx_buffer_used;

    if (_rx_buffer_size == _rx_buffer_used)
        if (_receiving)
            _rx_block.signal();
}

void TCP::Channel::push()
{
    if (_receiving)
        _rx_block.signal();
}

void TCP::Channel::closing()
{
    if (_receiving)
        _rx_block.signal();
}

void TCP::Channel::closed()
{
    if (_receiving)
        _rx_block.signal();

    if (_sending)
        _tx_block.signal();
}

void TCP::Channel::connected()
{
    _tx_block.signal();
}

void TCP::Channel::sent(u16 size)
```

```
{
    if (_sending) {
        _tx_bytes_sent += size;

        _tx_block.signal();
    }
}

void TCP::Channel::error(short errorcode)
{
    if (errorcode != ERR_TIMEOUT) {
        _error = errorcode;

        if (_receiving)
            _rx_block.signal();
    }

    if (_sending)
        _tx_block.signal();
}

void TCP::Channel::update(Data_Observed<IP::Address> *ob, long c,
                  IP::Address src, IP::Address dst,
                  void *data, unsigned int size)
{
        // TODO
}
__END_SYS
```

A.3 Classes utilitárias

A.3.1 buffer.h

```
#ifndef __buffer_h
#define __buffer_h

#include <system/types.h>
/*
 * Linked list of buffers
 * used to do zero-copy network send by stacking
 * the payload from multiple protocols:
 *
 * Example: EthernetPreamble + IP Packet + UDP Datagram + RTP Header + RTP Payload + EthernetCRC
 */
class SegmentedBuffer;

class SegmentedBuffer {
 protected:
        const char * _data;
        size_t _size;
        SegmentedBuffer * _next;
 public:
```

```
        SegmentedBuffer(const char * data,size_t size,SegmentedBuffer * next = 0) :
                _data(data), _size(size), _next(next) {}
        SegmentedBuffer(void * data,size_t size,SegmentedBuffer * next = 0) :
                        _data((const char *)data), _size(size), _next(next) {}

        void append(SegmentedBuffer * next) { _next = next; }

        const char * data() { return _data; }
        size_t size() { return _size; }
        SegmentedBuffer * next() { return _next; }

        size_t total_size() {
                return _size + (_next ? _next->total_size() : 0);
        }

        // method for final delivery (like copy to a DMA ring buffer)
        size_t copy_to(char * dst,size_t maxsize) {
                if (!_data)
                        return 0;

                if (_size < maxsize) {
                        memcpy(dst,_data,_size);
                        return _size + (_next ? (_next->copy_to(dst + _size,maxsize - _size)) : 0);

                } else {
                        memcpy(dst,_data,maxsize);
                        return maxsize;
                }
        }
};

#endif
```

A.3.2 observer.h

```
// EPOS Observer Utility Declarations

#ifndef __observer_h
#define __observer_h

#include <utility/list.h>

__BEGIN_SYS

// Observer
class Observer;

class Observed // Subject
{
    friend class Observer;

private:
    typedef Simple_List<Observer>::Element Element;

protected:
```

```
    Observed() {}

public:
    virtual ~Observed() {}

    virtual void attach(Observer * o);
    virtual void detach(Observer * o);
    virtual void notify();

private:
    Simple_List<Observer> _observers;
};

class Observer
{
    friend class Observed;

protected:
    Observer(): _link(this) {}

public:
    virtual ~Observer() {}

    virtual void update(Observed * o) = 0;

private:
    Observed::Element _link;
};


// Conditionally Observed
class Conditional_Observer;

class Conditionally_Observed // Subject
{
    friend class Conditional_Observer;

private:
    typedef
    List_Elements::Singly_Linked_Ordered<Conditional_Observer> Element;

public:
    Conditionally_Observed() {
        db<Observed>(TRC) << "Observed() => " << this << "\n";
    }

    virtual ~Conditionally_Observed() {
        db<Observed>(TRC) << "~Observed(this=" << this << ")\n";
    }

    virtual void attach(Conditional_Observer * o, int c);
    virtual void detach(Conditional_Observer * o, int c);
    virtual void notify(int c);

private:
    Simple_List<Conditional_Observer, Element> _observers;
};
```

```
class Conditional_Observer
{
    friend class Conditionally_Observed;

public:
    Conditional_Observer(): _link(this) {
        db<Observer>(TRC) << "Observer() => " << this << "\n";
    }

    virtual ~Conditional_Observer() {
        db<Observer>(TRC) << "~Observer(this=" << this << ")\n";
    }

    virtual void update(Conditionally_Observed * o, int c) = 0;

private:
    Conditionally_Observed::Element _link;
};

// Conditional Data Observer
// An Observer that receives useful data
template<typename T>
class Data_Observer;

template<typename T>
class Data_Observed // Subject
{
    friend class Data_Observer<T>;

public:
    typedef Data_Observer<T> Observer;
    typedef List_Elements::Singly_Linked_Ordered<Observer> Element;

    Data_Observed() {
        db<Observed>(TRC) << "Observed() => " << this << "\n";
    }

    virtual ~Data_Observed() {
        db<Observed>(TRC) << "~Observed(this=" << this << ")\n";
    }

    virtual void attach(Observer * o, long c) {
        db<Observed>(TRC) << "Observed::attach(o=" << o << ",c=" << c << ")\n";

            o->_link = Element(o, c);
            _observers.insert(&o->_link);
    }
    virtual void detach(Observer * o, long c) {
        db<Observed>(TRC) << "Observed::detach(obs=" << o << ",c=" << c << ")\n";

        _observers.remove(&o->_link);
    }
    virtual void notify(T src,T dst,long c,void * data,unsigned int size) {
        db<Observed>(TRC) << "Observed::notify(cond=" << c << ")\n";

        for(Element * e = _observers.head(); e; e = e->next()) {
            if(e->rank() == c) {
                db<Observed>(INF) << "Observed::notify(this=" << this
```

```
                                << ",obs=" << e->object() << ")\n";
                        e->object()->update(this, c, src, dst, data, size);
                }
        }

    }

    void count() { return _observers.size(); }

private:
    Simple_List<Observer, Element> _observers;
};

template<typename T>
class Data_Observer
{
public:
    friend class Data_Observed<T>;
    typedef Data_Observed<T> Observed;

    Data_Observer(): _link(this) {
        db<Observer>(TRC) << "Observer() => " << this << "\n";
    }

    virtual ~Data_Observer() {
        db<Observer>(TRC) << "~Observer(this=" << this << ")\n";
    }

    virtual void update(Observed * o, long c,T src,T dst,void * data,unsigned int size) = 0;

private:
    typename Observed::Element _link;
};

__END_SYS

#endif
```

A.4 Testes

A.4.1 icmp_test.cc

```
#include <icmp.h>
#include <alarm.h>

__USING_SYS

/*
 * Simple PING test application
 */
OStream cout;
```

```
class ReplyObserver : public ICMP::Observer
{
public:
        void update(ICMP::Observed * ob,long type,IP::Address from,IP::Address to,
                    void* data,unsigned int size) {
                ICMP::Packet * pkt = (ICMP::Packet*)data;
                cout << "Echo reply received from " << from
                     << " sequence number " << pkt->sequence() << endl;
        }
};

int main()
{
        IP ip(0);

        //QEMU IP settings
        ip.set_address(IP::Address(10,0,2,15));
        ip.set_gateway(IP::Address(10,0,2,2));
        ip.set_netmask(IP::Address(255,255,255,0));


        ICMP icmp(&ip);

        ReplyObserver obs;
        icmp.attach(&obs, ICMP::ECHO_REPLY);

        for (int seq=0;seq < 4;++seq)
        {
                ICMP::Packet pkt(ICMP::ECHO,0,123123,seq);
                icmp.send(IP::Address(10,0,2,2),pkt);
                Alarm::delay(100000);
        }
}
```

A.4.2 dhcp_test.cc

```
#include <dhcp.h>
#include <alarm.h>

__USING_SYS

OStream cout;

int main()
{
    IP ip(0);

    ip.set_address(IP::NULL);
    ip.set_gateway(IP::NULL);
    ip.set_netmask(IP::NULL);

    DHCP::Client dhcpc;

    dhcpc.configure();
```

```
    Alarm::delay(5000000);

    cout << "IP Addr: " << dhcpc.address() << endl;
    cout << "Netmask: " << dhcpc.netmask() << endl;
    cout << "Gateway: " << dhcpc.gateway() << endl;
    cout << "DNS:  " << dhcpc.nameserver() << endl;

    if (dhcpc.address() == IP::NULL)
        cout << "@result = failed\n";
    else
        cout << "@result = passed\n";
}
```

A.4.3 tcp_test.cc

```
#include <tcp.h>
#include <utility/string.h>
#include <mutex.h>

__USING_SYS

OStream cout;

class HTTPServer : public TCP::ServerSocket {
public:
    HTTPServer() : TCP::ServerSocket(TCP::Address(tcp()->ip()->address(),80)) {}

    TCP::Socket* incoming(const TCP::Address& from) {
        // we can clone here to accept multiple connections
        // or just return itself
        return this;
    }

    void connected() {
        cout << "Connection from " << remote() << endl;
    }

    void closed() {
        cout << "Disconnected from " << remote() << endl;

        listen();
    }

    void error(short err) {
        cout << "Connection error" << endl;
    }

    void sent(u16 size) {}

    void received(const char *data,u16 size) {
        cout << "Received "<<size<<" bytes: " << data << endl;

        const char * msg = "200 HTTP 1.1\r\n\r\nHello world!\r\n";
        send(msg, 30);
        close();
```

```
    }
};

class WebClient : public TCP::ClientSocket {
public:
    WebClient() :
        TCP::ClientSocket(
                TCP::Address("74.125.234.84:80"),
                TCP::Address(tcp()->ip()->address(),55000 + Pseudo_Random::random() % 10000))
    {                                                                                    50
        m.lock();
    }

    void send_request() {
        send("GET / HTTP/1.1\n\rHost: www.google.com\n\r\n\r",40);
    }
    void connected() {
        cout << "Connected to " << remote() << endl;
        send_request();
    }                                                                                    60

    void closing() {
        close();
    }

    void closed() {
        cout << "Disconnected from " << remote() << endl;
        m.unlock();
    }
                                                                                         70
    void error(short err) {
        if (err == ERR_TIMEOUT) {
            cout << "timeout occured\n";
            switch (state()) {
                case SYN_SENT:
                    connect();
                    break;
                case ESTABLISHED:
                    send_request();
                    break;                                                               80
                default:
                    abort();
                    m.unlock();
            }
        }
        else {
            cout << "Connection error" << endl;
            m.unlock();
        }
    }                                                                                    90

    void sent(u16 size) {
        cout << "Bytes sent: " << size << endl;
        close();
    }

    void received(const char *data,u16 size) {
        cout << "Received "<<size<<" bytes: " << endl;
```

```
        int p;
        for(p=0;p<size;p++)
            cout << *data++;
    }

    void wait() { m.lock(); }
protected:
    Mutex m;
};

int main()
{
    IP * ip = IP::instance();

    ip->set_address(IP::Address(10,0,2,15));
    ip->set_gateway(IP::Address(10,0,2,2));
    ip->set_netmask(IP::Address(255,255,255,0));

    //HTTPServer httpd(&tcp);
    //Thread::self()->suspend();

    WebClient web;

    web.wait();
    delete ip; // kill IP thread
}
```

A.4.4 tcp_channel_test.cc

```
#include <tcp.h>
#include <utility/string.h>

__USING_SYS

OStream cout;

void web_client() {

    TCP::Channel channel;

    channel.bind(55000 + Pseudo_Random::random() % 10000);

    if (!channel.connect(TCP::Address("74.125.234.84:80")))
    {
        cout << "Connection failed\n!";
        return;
    }


    channel.send("GET / HTTP/1.1\n\rHost: www.google.com\n\r\n\r",40);

    char * data = new char[4096];

    int size = channel.receive(data,4096);
```

```
    cout << "Received "<<size<<" bytes: " << endl;
    int p;
    for(p=0;p<size;p++)
        cout << *data++;

    channel.close();
};

int main()
{
    IP * ip = IP::instance();

    ip->set_address(IP::Address(10,0,2,15));
    ip->set_gateway(IP::Address(10,0,2,2));
    ip->set_netmask(IP::Address(255,255,255,0));

    web_client();

    delete ip; // kill IP thread
}
```

Interoperabilidade de sistemas embarcados com uma pilha de protocolos flexível, configurável e de baixo custo

Rodrigo Valceli Raimundo[1]

[1]Universidade Federal de Santa Catarina (UFSC)

rodrigovr@lisha.ufsc.br

Resumo. *A tendencia atual da Internet é permear todos os objetos e interações da vida cotidiana. Embora a conectividade com a grande rede seja assunto resolvido para uma vasta gama de sistemas computacionais, temos nichos crescentes de objetos do dia a dia que estão adquirindo maiores funcionalidades em virtude do avanço constante da eletrônica embarcada. Neste trabalho apresentamos nossa solução de conectividade IP desenvolvida na plataforma EPOS, demonstrando a viabilidade de interconexão usando protocolos maduros e bem estabelecidos mesmo em sistemas altamente reduzidos e especializados.*

Abstract. *The Internet's current trend is to permeate all objects and interactions of everyday life. Although the connectivity to the Internet is a finished subject for a wide range of computer systems, we have growing niches of everyday objects that are getting more functionality due to the constant advancement of embedded electronics. We present our solution for IP connectivity in our EPOS development platform, demonstrating the feasibility of using mature and well established interconnection protocols even in highly reduced and specialized systems.*

1. Introdução

Comunicação de dados é um dos pilares da Ciência da Computação e a pesquisa nesta área, como disciplina científica, nos remete no mínimo à invenção do telégrafo. Porém não é por ser um campo de pesquisa antigo que seus assuntos estejam esgotados. Há pelo menos trinta anos vivemos uma explosão na dimensão e alcance dos sistemas de telecomunicação, com um forte destaque para a Internet. O que no final da década de 70 se limitava aos poucos (e enormes) computadores das principais universidades dos Estados Unidos, hoje está presente em uma parcela significativa dos lares, empresas e demais organizações. Aquilo que começou com o intuito de sobreviver a grandes calamidades hoje é tão onipresente e complexo que até mesmo os especialistas tem dificuldade em definir suas dimensões.

Em paralelo ao crescimento explosivo das redes de computadores também houve decréscimo exponencial no custo e tamanho do computadores. Chegamos na fronteira tecnológica da revolução dos circuítos integrados de larga escala e alta densidade. Tal redução levou ao surgimento dos denominados sistemas embarcados, algumas vezes chamados de sistemas embutidos ou dedicados. Muitos são os objetivos dos sistemas embarcados atuais e talvez a maior semelhança entre todos eles é de que as pessoas não os veem

como um computador no seu sentido convencional, como as estações de trabalho fixas e portáteis. O encolhimento dos dispositivos computacionais e seu barateamento também possibilitou a inserção de um computador onde antes não se imaginava que fosse possível e/ou necessário. Quem alguns anos atrás acharia viável colocar um computador em cada tomada, lâmpada, ar condicionado, vestuário e até mesmo dentro de nosso corpo? Hoje a ideia não soa tão absurda assim.

A união dessas duas revoluções alimenta nossa pesquisa atual de diversas maneiras. Atualmente temos pesquisa de ponta em MANETs (Redes de agentes móveis) e WSNs (Redes de sensores sem-fio), por exemplo. Porém o rumo atual nos leva a uma ideia muito mais abrangente. Interconectar todos os objetos do mundo criaria o que chamamos de IoT (Internet das Coisas, do inglês Internet of Things). Alguns já pensam um pouco mais além, na Web of Things (Web das Coisas). A estrutura de software e hardware desenvolvida para a Internet de super-computadores e desktops se mostrou muito custosa e inapropriada para os novos sistemas computacionais extremamente reduzidos e com diversas restrições que não se aplicavam aos sistemas computacionais tradicionais como: consumo de energia, dimensões e custo. Tais limitações fizeram surgir diversos protocolos e implementações voltados para esse novo mundo.

A união dos eventos acima, dentro do contexto do LISHA (Laboratório de Integração Software/Hardware) da UFSC e cujo projeto de longo prazo EPOS (Embedded Parallel Operating System) serve como base para nossa pesquisa em redes e sistemas embarcados, levou ao desenvolvimento do presente trabalho. O objetivo central deste trabalho é projetar e desenvolver a infra-estrutura necessária de comunicação para que dispositivos embarcados com aplicações desenvolvidas com o EPOS possam se comunicar com os demais dispositivos da Internet. Para cumprir tal objetivo é preciso aderir às normas já estabelecidas de comunicação para Internet, a famosa pilha de protocolos TCP/IP, provendo os recursos necessários para o desenvolvimento de aplicações com mínimo custo de recursos computacionais, como uso de CPU, memória e tamanho de código, além de portabilidade entre as diversas arquiteturas de hardware disponíveis no mercado.

Este trabalho está organizado da seguinte maneira. A sessão inicial faz uma revisão do estado da arte e enumera os principais conceitos envolvidos ao longo do texto. A seguinte mostra a arquitetura desenvolvida em detalhes. Nos resultados temos uma análise comparativa entre nosso projeto e outros relacionados. Por fim temos as conclusões e ideias para trabalhos futuros.

2. Estado da Arte

A pilha de protocolos TCP/IP se consolidou como o padrão *de-facto* para comunicação de dados (DUNKELS et al., 2004), atingindo atualmente áreas já então consolidadas antes do surgimento da Internet, como a telefonia (VoIP) e a televisão (IP-TV) (GOODE, 2002). A pesquisa em intercomunicação IP no mundo dos sistemas embarcados é recente e ainda encontra muitos desafios, porém acredita-se que o futuro está na interconexão de todos os objetos do dia-a-dia (VASSEUR; DUNKELS, 2010). O objetivo desse capítulo é levantar os modelos e implementações com enfase nesta categoria de sistemas. A literatura explicando os detalhes dos protocolos TCP/IP é extensa e ficaria muito massante explicar tais conceitos básicos neste trabalho. Caso necessário uma abordagem didática pode ser encontrada em Tanenbaum (2002).

2.1. TCP/IP em sistemas convencionais

A história de praticamente todas as implementações do protocolo TCP/IP para sistemas convencionais remota a história do BSD, o sistema Unix melhorado e redistribuído pela Universidade da Califórnia). Um fato histórico importante é de que, o código Unix cedido pela AT&T possuía um alto custo de licenciamento, mas o código responsável pelos protocolos de rede foram desenvolvidos completamente por Berkeley e distribuídos por uma licença de software livre. Tal disponibilidade de uma implementação de qualidade e livre fez com que os demais produtores de sistemas operacionais adotassem a implementação oriunda do BSD. O efeito prático disto foi uma homogenização da API e uma rápida difusão do TCP/IP frente aos concorrentes de sua época.

As duas primeiras implementações BSD (TCP-Tahoe e TCP-Reno) focaram em controle de congestionamento. Seus descendentes modernos como TCP CUBIC (HA; RHEE; XU, 2008) utilizado no Linux e o Coumpond TCP (TAN; SONG, 2006) utilizado a partir do Windows Vista, focaram em melhorias como *fairness* e aproveitamento máximo de banda disponível. Outras implementações como TCP SACK e TCP Fast Reno também alcançaram boa popularidade. O trabalho publicado por Mascolo (2006) mostra que tais mecanismos de controle de congestionamento podem ser descritos a partir de pequenas modificações em um mesmo modelo matemático. Uma comparação entre New Reno, CUBIC e Compound feita por Abdeljaouad et al. (2010) mostra resultados parecidos de desempenho tanto para conexões cabeadas quanto para conexões sem-fio entre as diversas implementações.

2.2. TCP/IP para sistemas embarcados

2.2.1. uIP e lwIP

Com o intuito de permitir conectividade IP a microcontroladores surgiu a primeira pilha TCP/IP para arquiteturas de 8-bit, chamada de uIP (micro-IP) e apresentada no trabalho de Dunkels (2003). Para a implementação do uIP se concretizar foram aplicadas várias limitações para minimizar o uso de memória, entra elas podemos citar:

1. Nenhuma alocação dinâmica, número máximo de conexões limitado em tempo de compilação,
2. *Buffer* de recebimento e envio único e compartilhado para ambas operações,
3. Sem demultiplexação automática,
4. Retransmissão assistida pela aplicação,
5. Sem controle de fluxo, apenas um segmento enviado por vez,
6. Suporte a uma única interface de rede,
7. Implementação monolítica desde o encapsulamento da camada de enlace até a camada de aplicação.

Tais limitações fizeram o uIP alcançar um tamanho total o pequeno suficiente para rodar em microcontroladores com 8kb de ROM. O autor afirma ainda que a portabilidade é alta, já que o código é C-puro e a interface externa necessita apenas de operações básicas de *send/receive* em uma interface de rede e um temporizador para retransmissões. Por outro lado, o esforço do lado do desenvolvedor de aplicações é maior que em outras implementações da pilha TCP/IP. Lidar com múltiplas conexões ou protocolos distintos,

como UDP e TCP simultaneamente, se torna bastante complicado com a API limitada oferecida pelo uIP.

Outro esforço do mesmo autor foi a criação do lwIP (lightweight-IP), publicada no mesmo trabalho (DUNKELS, 2003) e presente no sistema operacional Contiki (DUNKELS; GRÖNVALL; VOIGT, 2004). A lwIP resolve a maior parte das limitações do uIP, com um consumo de recursos um pouco maior. A API da lwIP, embora lembre a implementação tradicional de *sockets* do Unix BSD, é feita utilizando o conceito de *protothreads*, também do mesmo autor (DUNKELS; SCHMIDT, 2005). As *protothreads* ajudam a programação orientada a eventos no Contiki, permitindo expressar de forma sequencial procedimentos que são na realidade passos de uma máquina de estados. Tal abordagem é bem vinda no contexto do Contiki, já que o paradigma do mesmo é orientação a eventos e ele não possui suporte nativo a threads.

2.2.2. lyraNET

Algumas aplicações embarcadas, com menos restrições de memória, podem optar por executar sistemas operacionais de propósito geral, como o Linux ou o uCLinux. A lyraNet (LI; CHIANG, 2005) foi desenvolvida como uma alternativa a pilha TCP/IP do Linux, com o objetivo de reduzir ao mínimo o número de operações de cópia de memória. Tal objetivo é justificado pelo fato da maior parte do tempo de processamento das operações de rede serem gastos em operações de cópia entre a memória de kernel e a memória de usuário.

A lyraNET consegue ao mesmo tempo oferecer tanto um desempenho melhor (tempo de execução), quanto um *overhead* menor de memória em relação pilha tradicional do Linux. Embora do ponto de vista da aplicação nada seja alterado, a adapção da lyraNET para uma nova plataforma requer também a adaptação dos drivers das interfaces de rede.

2.2.3. nanoIP, nanoTCP e nanoSLP

Embora possuam IP e TCP no nome, a pilha de protocolos nanoIP (SHELBY et al., 2003) não respeita os formatos de cabeçalho definidos para os protocolos da Internet (POSTEL, 1981a, 1981b, 1980). O objetivo da pilha nanoIP foi levar alguns recursos do TCP/IP convencional para redes de sensores ou outros sistemas reduzidos, principalmente os que usam o padrão IEEE 802.15.4 para comunicação sem-fio. O nanoIP não suporta roteamento diretamente, apenas com a ajuda de *gateways* especializados. A noção de endereçamento da camada de rede foi eliminada, sendo utilizado apenas o endereçamento da camada de enlace. Todas as modificações e reduções impostas nos cabeçalhos foram para diminuir o peso do nanoIP nos pacotes IEEE 802.15.4, que transporta no máximo 127 octetos por pacote.

Um ponto interessante da proposto do nanoIP, foi o nanoSLP. O nanoSLP é um protocolo de localização de serviços baseado em consultas textuais. Sua criação foi uma tentativa dos autores de conseguirem algo análogo ao *DNS* (MOCKAPETRIS, 1987) e aos *Directory Services*, como o LDAP (HOWES; SMITH, 1995), para as redes *mesh* construídas com nanoIP.

2.2.4. Implementações proprietárias

Além das pilhas TCP/IP citadas acima, existe um mercado de implementações proprietárias (algumas gratuitas, outras não) consolidado. A situação mais recorrente neste cenário é o fornecimento por parte dos fabricantes de microcontroladores, como a Texas Instruments, de kits de desenvolvimento de software contendo pilhas TCP/IP completas. Além do NDK (Network Development Kit) da TI, podemos citar o uC/TCP da Micrium, o MISRA TCP/IP da empresa HCC Embedded, IPLite da Interpeak, entre outros. Devido as restrições de disponibilidade destas soluções não é possível fazer uma boa análise apenas baseada em *overviews* disponibilizados pelos fabricantes. O fato de existirem diversas ofertas de soluções pagas ajuda a consolidar a importância do nicho de aplicação explorado neste trabalho.

3. Arquitetura

Este capítulo aborda as decisões tomadas na elaboração e desenvolvimento de uma pilha TCP/IP para o EPOS. Incluindo a interação com os demais componentes do sistema, a organização interna, utilitários desenvolvidos e interface de programação para os desenvolvedores de aplicativos.

3.1. Visão Geral

Atualmente dois modelos distintos de programação são amplamente difundidos para a implementação de APIs/frameworks de comunicação de dados. O primeiro é o modelo orientado a eventos, um modelo de computação reativo onde o usuário (programador) da infraestrutura de comunicação interage com o meio de comunicação escrevendo procedimentos conhecidos como *callbacks*. Tais callbacks são executados automaticamente pelo sistema quando um evento associado com eles for disparado. No caso de comunicação, eventos comuns são a chegada de dados, o estabelecimento e término de conexões, por exemplo. O modelo orientado a eventos está mais próximo do paradigma descritivo de programação, já que o programador diz o que deve ser executado, mas não sabe necessariamente quando será executado. De outro lado, o segundo modelo, síncrono/sequencial está relacionado com a programação procedural e de certa forma é mais difundido e utilizado. Uma API de comunicação síncrona possui funções de envio/recebimento, estabelecimento de conexão e afins, que são usadas pelo programador de maneira sequencial dentro do fluxo da aplicação.

Neste trabalho ambos os modelos, orientado a eventos e sequencial, foram usados para construir uma interface de programação que possa ser utilizada de acordo com as necessidades da aplicação, seguindo a metodologia ADESD (Application-Driven Embedded Software Design) (FRÖHLICH, 2001).

3.1.1. Descrição do modelo orientado a eventos/assíncrono

A hierarquia de classes do modelo orientado a eventos está exibida na figura 1. No topo da hierarquia temos a classe NIC, que representa a interface de rede e em ADESD, é classificada como um mediador de hardware (POLPETA; FRÖHLICH, 2004). O objetivo

do mediador é fornecer uma interface homogênea para utilização de dispositivos de hardware. Diferente dos tradicionais *device drivers*, o mediador é construído de forma que seu código possa ser incorporado pelos demais componentes do sistema sem a necessidade de interfaces pesadas como as *syscalls* utilizadas em boa parte dos sistemas operacionais.

Após a classe NIC, temos as classes intermediárias da infraestrutura: IP, UDP, ICMP e TCP. A divisão dos diversos elementos da camada de rede e camada de transporte em classes distintas foi feita para aumentar o desacoplamento, permitindo que os componentes possam ser usados de acordo com sua necessidade, sem apresentar adição de código não-usado no binário final de uma aplicação. Tal separação também permite um melhor entendimento, depuração e separação de conceitos dentro do código-fonte. A ligação entre as classes das diversas camadas é feita utilizando o padrão de projeto *Publish/Subscribe* (EUGSTER et al., 2003), fornecido pelas classes utilitárias Conditional_Observer, Conditional_Observer, Data_Observer e Data_Observed, apresentadas nos diagramas da figura 2.

Na ponta da hierarquia, temos as classes UDP::Socket e TCP::Socket, que devem ser utilizadas pelo desenvolvedor das aplicações através de herança, implementados os métodos necessários para recepção de dados e controle de conexão no caso do TCP::Socket.

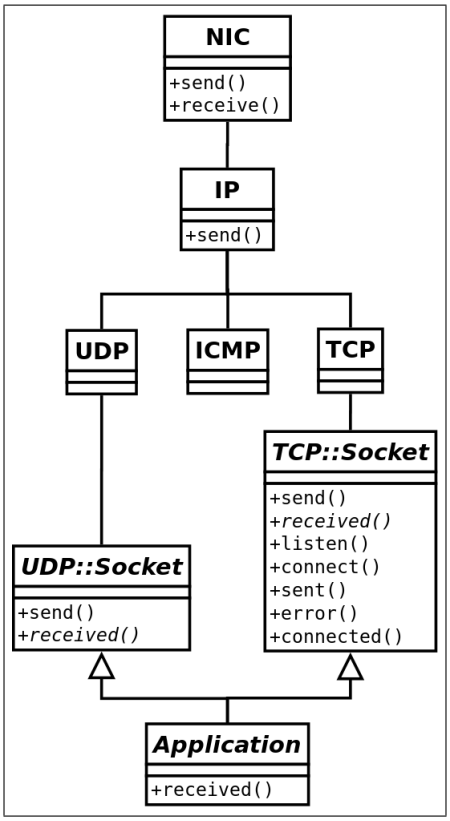

Figura 1. Diagrama de classes simples do modelo orientado a eventos

Vantagens e desvantegens A API orientada a eventos deste trabalho permite a comunicação utilizando UDP e TCP com mínimo *overhead*, tanto de código, quanto de memória e tempo de CPU. Tal desempenho porém é sacrificado pelo fato de que o desenvolvimento

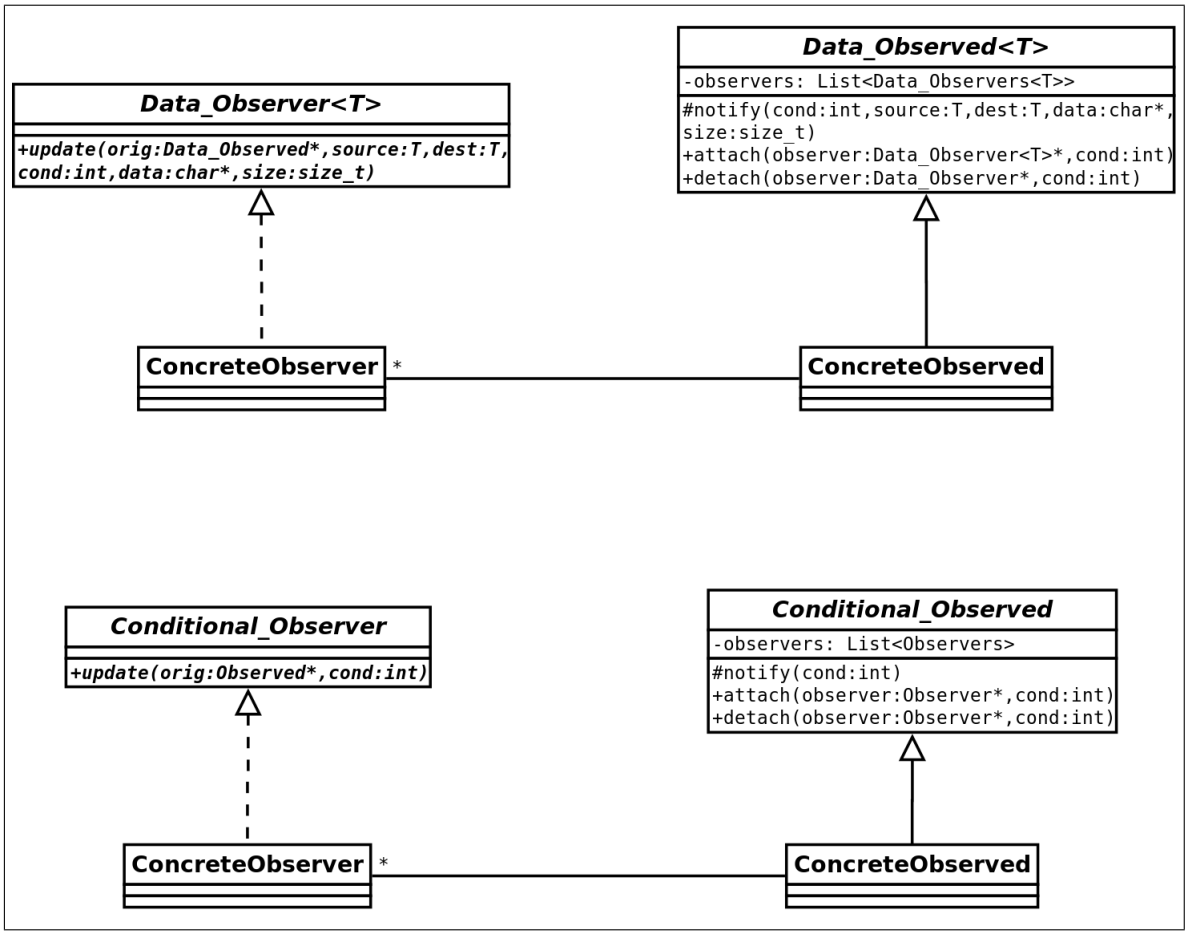

Figura 2. Diagrama das classes relacionadas ao padrão Publish/Subscribe

de aplicações é significativamente mais difícil. Nossa implementação orientada a eventos segue um modelo parecido com o proposto por Dunkels (2009), onde as retransmissões devem ser assistidas pela aplicação.

3.1.2. Descrição do modelo sequencial/síncrono

O modelo sequencial, visto na figura 3, foi projetado como uma evolução do modelo orientado a eventos. Na API sequencial temos duas novas classes, UDP::Channel e TCP::Channel. Tais classes implementam os *callbacks* necessários das classes TCP::Socket e UDP::Socket e proveem uma interface de programação linear que se assemelha às interfaces modernas de linguagens como Java e C#. Na classe UDP::Channel, por exemplo, temos no lugar do callback received() o método receive(), que quando executado, bloqueia até que uma mensagem seja rececebida. Já no caso do TCP::Channel, temos a vantagem que todo controle de timeout e retransmissões é feito sem nenhuma intervenção do desenvolvedor da aplicação. Como visto na parte inferior da figura 3, o modo de utilização por parte da aplicação deixa de ser através do mecanismo de herança e passa a ser através de associação/composição.

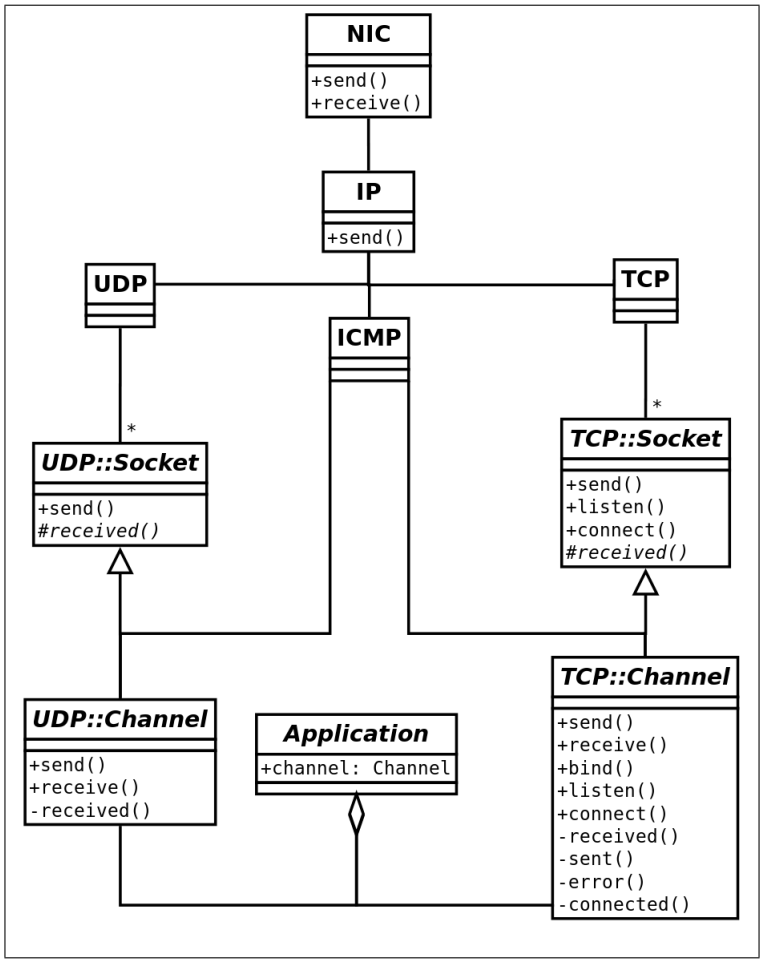

Figura 3. Diagrama de classes simples do modelo síncrono

Vantagens e desvantegens O modelo sequencial é o mais simples de ser utilizado, possuindo semântica equivalente a tradicional API de *Sockets* do BSD, difundida entre os demais sistemas operacionais. A utilização de memória, tanto para código, quanto em tempo de execução, porém, é maior neste caso. O que pode, em algumas circunstancias inviabilizar sua utilização.

3.2. Modelo e Comportamento

O objetivo dessa seção é analisar melhor a implementação de cada aspecto das diversas camadas de software que compõe este trabalho. Como houve desde o início a preocupação entre relacionar diretamente as camadas do modelo TCP/IP diretamente com classes em um modelo orientado a objetos, também houve a preocupação em fazer com que cada classe (ou camada) utilize diretamente a camada inferior e forneça serviços à camada superior, porém sem depender desta, assim como proposto no modelo de referência OSI. Tal aproximação entre uma implementação prática e o modelo teórico permite a análise separada de cada componente da implementação, facilitando a depuração de eventuais problemas e até mesmo o estudo para fins didáticos.

3.2.1. Camada de rede

A camada central deste projeto é a camada de rede, representada pela classe IP. De uma maneira geral, podemos dizer que cada objeto IP funciona como um ponto central de coordenação. O padrão de projetos *Active Object* foi escolhido como base para a classe IP pelos seguintes motivos:

1. O EPOS possui uma implementação de *threading* sofisticada, logo o confinamento do processamento de pacotes IP dentro de uma *thread* permite que políticas de escalonamento sejam aplicadas. Se todo processamento necessário para recepção de pacotes IP fosse feito dentro de um contexto de tratamento de interrupções, a utilização de IP junto a um sistema de tempo real se tornaria quase impossível.
2. Nem toda interface de rede é dotada de mecanismos de sinalização assíncronos, como as interrupções. Logo a necessidade de utilizar o mecanismo de *polling* faz com que o recebimento de pacotes seja uma tarefa ativa e não uma ISR (*Interrupt Service Routine*).

As duas principais tarefas da camada IP são descritas de uma maneira simples pelos diagramas de atividade das figuras 4 e 5. O comportamento básico da recepção consiste em verificar consistências e notificar a camada superior (utilizando o padrão Publish/Subscribe mencionado anteriormente). Já no envio de pacotes, a camada IP é responsável principalmente pelo encapsulamento correto dos dados (incluindo *checksums*). A descoberta de endereços físicos a partir de endereços lógicos é feita separadamente por uma classe chamada Network_Service. Dependendo do cenário de execução e das configurações que o desenvolvedor escolher para o sistema o Network_Service será implementado por uma classe distinta. Atualmente temos três Network_Service's distintos:

1. *ARP_Service:* Resolve endereços físicos utilizando o protocolo ARP. Útil para redes Ethernet.
2. *BCast_Service:* Envia todos os pacotes para o endereço de *broadcast*. Útil para conexões ponto-a-ponto e redes sem-frio.
3. *ADHOP_Service:* Utiliza o protocolo ADHOP (OKAZAKI; FRÖHLICH, 2009, 2011) para determinar o próximo salto. Útil para redes sem-fio multi-hop.

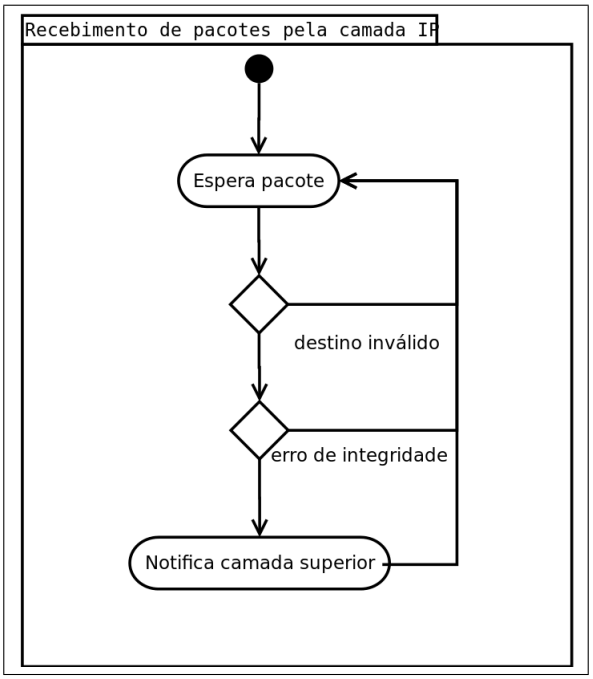

Figura 4. Diagrama de atividade do recebimento de pacotes pela camada de rede.

3.2.2. Camada de transporte: UDP

Atualmente apenas dois protocolos da camada de transporte são largamente utilizados, sendo eles o UDP e o TCP. Outros protocolos como o SCTP e DCCP (ONG; YOAKUM, 2002; KENT; ATKINSON, 1998) não são difundidos e por isso não foram contemplados neste trabalho.

Comparado ao TCP, o UDP é extremamente simples e por isso sua interface de software também é bastante reduzida. O principal recurso oferecido pelo UDP é a multiplexação, ou endereçamento a nível de aplicação, que permite que vários agentes (aplicações por exemplo) utilizem o mesmo endereço de rede para fluxos de comunicação distintos, sem que cada agente precise ter conhecimento da existência dos demais utilizadores da rede. De uma maneira geral, o UDP é amplamente utilizado nos serviços de DNS, DHCP e transmissões multimídia. Cada um destes três tipos de serviço faz uso de uma funcionalidade (ou falta de) do UDP.

Para o DNS, a falta de estabelecimento de conexão é ideal, já que a consulta a nomes costuma ser composta de apenas duas mensagens distintas, uma de consulta e outra de resposta, assim sendo o estabelecimento e finalização de uma conexão, dobrariam a quantidade mínima necessária de troca de mensagens.

O DHCP, por sua vez, utilizado para configuração automática de atributos de um *host* dentro de uma rede, faz uso da capacidade do UDP de permitir endereçamento do tipo um-para-muitos (*broadcast*). Tal endereçamento se faz necessário pois no momento da autoconfiguração o *host* não possui nenhuma informação da rede e por isso não pode endereçar nenhum destino com precisão.

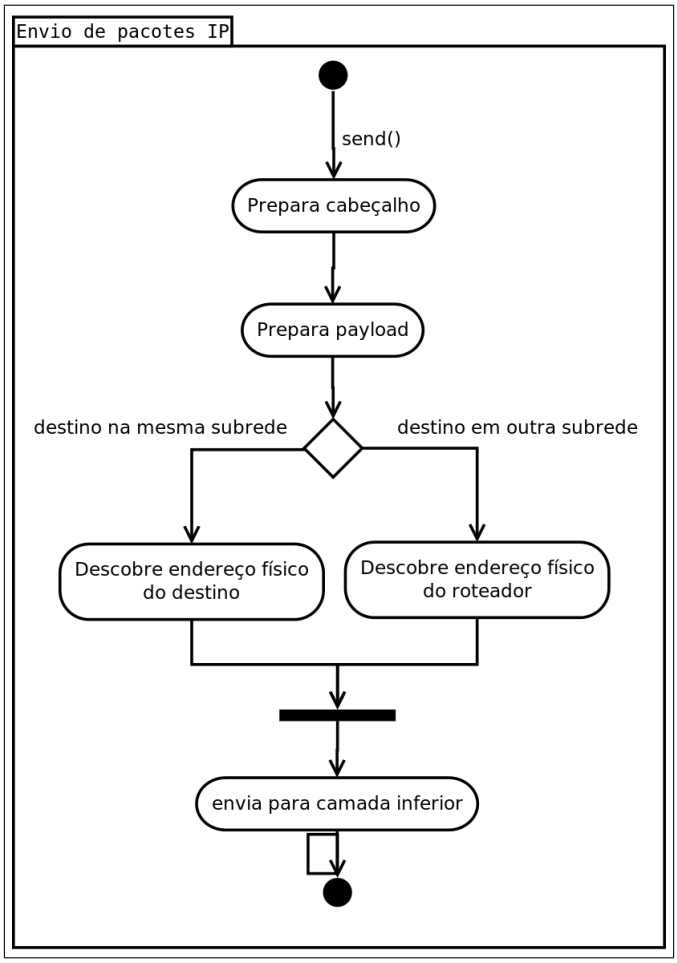

Figura 5. Diagrama de atividade do envio de pacotes pela camada de rede.

Os serviços multimídia, principalmente os de tempo-real ou baixa latência, tiram proveito da ausência de confirmação e retransmissão inerente ao UDP. Para a percepção humana, a perda de alguma informação de áudio/vídeo é menos importante ao entendimento do que o atraso ou a variação do mesmo (*jitter*).

Nossa implementação do UDP foi planejada de maneira a contemplar os cenários de utilização mencionados acima, com foco central em praticidade para o desenvolvedor. As figuras 6 e 7 esquematizam a estrutura das classes utilizas para comunicação UDP nos dois modelos desenvolvidos neste trabalho. Vale lembrar que algumas operações, como por exemplo a construção de instancias de UDP::Address a partir de representações textuais foram omitidas do diagrama para melhor entendimento.

3.3. Camada de transporte: TCP

3.3.1. Alterações no TCP

Neste trabalho algumas características ou funcionalidades originais do TCP não foram implementadas por motivos técnicos. Na sequência serão expostos tais itens e o porquê da omissão de cada um.

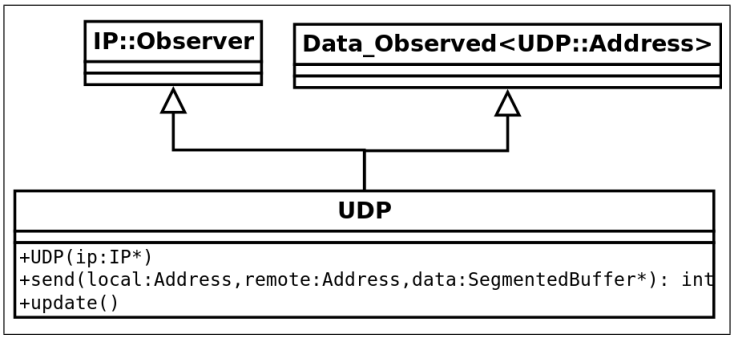

Figura 6. Esqueleto da classe UDP.

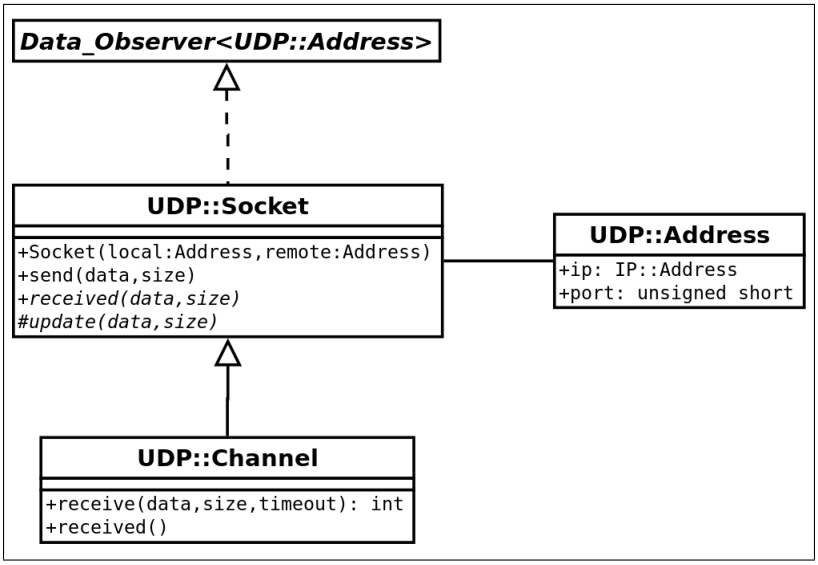

Figura 7. Interface de programação UDP::Socket e UDP::Channel.

Urgent Pointer A recomendação feita na seção 5 de Gont e Yourtchenko (2011) é para que novas aplicações não utilizem o mecanismo de Urgent Point do TCP. Os principais motivos são os problemas semânticos decorridos da implementação independente por diversos fabricantes. Na prática observa-se que praticamente nenhuma aplicação depende do TCP Urgent Point para o correto funcionamento. Logo, ignorar o mecanismo em questão não implica em nenhuma perda de utilidade para os cenários típicos em que esta implementação será utilizada.

Recebimento de dados no estado CLOSE_WAIT A operação **CLOSE** tradicionalmente implementada fecha apenas uma das pontas da conexão, permitindo ainda o recebimento de dados enquanto a outra ponta não executa a mesma operação. Para simplificar a semântica e manter um interface de programação mais simples, o método **close** da classe TCP::Channel foi projetado de modo a anunciar uma janela de tamanho zero e esperar o encerramento síncrono da conexão, não permitindo a utilização do método **receive** após seu uso. Na implementação orientada a eventos tal limitação não foi necessária, alem disso a classe TCP::Socket possui um callback chamado **closing** que pode ser utilizado para saber que a outra ponta fechou seu fluxo.

Controle de congestionamento Tal como no uIP, o controle de congestionamento desta implementação foi praticamente eliminado. As técnicas aplicadas no TCP-Reno e seus sucessores têm por objetivo a saturação da rede para melhor aproveitamento da banda e encaram perdas de pacote como sinal de congestionamento. Tal propriedade não é válida para todas as topologias de rede, como o IEEE 802.11.4 ou redes PLC. Outra característica associada ao controle de congestionamento é o *Delayed Ack*, sendo que este não foi implementado pois sua utilização junto ao nosso modelo de janela zero, que será explicando adiante, reduziria ainda mais o aproveitamento da banda de transmissão.

3.3.2. TCP com "Janela Zero"

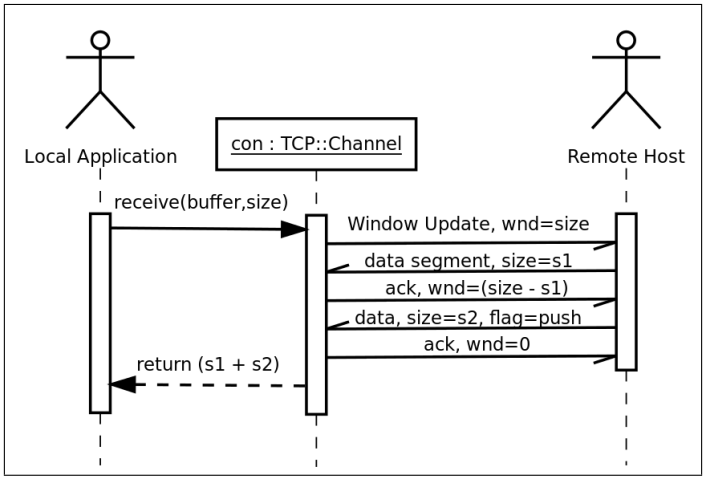

Figura 8. Recebimento de dados no TCP com janela zero

O controle de fluxo nas conexões TCP é feito através da janela de recebimento. A janela de recebimento é um campo presente em todos os seguimentos TCP que indica quantos bytes o remetente está disposto a receber. No uIP a janela anunciada é igual ao MSS, sendo que tal característica advém da natureza orientada a eventos do uIP e da utilização de um único buffer universal para envio e recebimento. Por outro lado, o tamanho da janela TCP em sistemas convencionais costuma ser alto e técnicas como a utilização de TCP-Options para aumentá-la além dos limites originais são utilizadas. A quantidade de memória disponível em sistemas convencionais é várias ordens de magnitude superior ao necessário para o aproveitamento máxima da largura de banda pelo TCP e logo limitar o tamanho da janela se torna dispensável.

Levando os dois exemplos acima em consideração, nossa arquitetura abraça as duas propostas. Ao utilizar a API orientada a eventos a janela anunciada é sempre de um segmento, enquanto ao utilizar a API síncrona a janela anunciada passar a representar o buffer de recebimento fornecido pela aplicação, tal como representado no diagrama 8. O comportamento final, o qual damos o nome de TCP com Janela Zero, passa a ser caracterizado por dois estados:

- Um estado **idle** com janela de tamanho zero, quando a aplicação não está disposta a receber dados,
- Um estado **ativo** quando a aplicação executa o método **receive** em um Channel e a janela anunciada passa a ser o buffer fornecido pela aplicação.

Na operação de envio, a aplicação também possui um papel central no comportamento da camada de transporte. Novamente nossa implementação se equilibra entre a simplicidade do uIP e a eficiência dos grandes sistemas operacionais. No caso o uIP, todas as retransmissões são responsabilidades do desenvolvedor das aplicações, cabendo ao uIP apenas alertá-lo sobre os *timeouts* ocorridos. Em sistemas convencionais, o nível de abstração é alto e o desenvolvedor fica completamente isolado de detalhes como retransmissão. Novamente, cada uma das duas APIs desenvolvidas se aproxima dos dois lados da questão:

- Utilizando TCP::Socket a aplicação pode utilizar deliberadamente a operação **send**, que retorna imediatamente. A cada segmento confirmado, o callback **sent** é executado para informar a aplicação quantos bytes a outra ponta recebeu. Em caso de timeout, o callback **error** com o parâmetro ERR_TIMEOUT é utilizado para informar a aplicação do problema. Neste cenário a aplicação decide como proceder com a retransmissão.
- A classe TCP::Channel, por outro lado, fornece uma operação **send** que bloqueia até que todos os dados tenham seu recebimento confirmado ou algum erro aconteça. Neste caso a aplicação fica isolada do problema das retransmissões, porém a utilização da banda depende diretamente do tamanho dos dados fornecido pela aplicação, quanto maior o buffer, maior o aproveitamento.

4. Resultados

Como visto nas tabelas 1 e 2, a diferença de tamanho entre o código gerado é mínima para a arquitetura Intel x86, com uma uma interface de rede Ethernet e utilizando ARP, e para a arquitetura ARM, no caso o EPOS Mote II, com C-MAC em uma rede 802.15.4 e roteamento ADHOP. A principal diferença encontrada no tamanho do componente IP, que ficou maior no ARM, deve-se ao fato da classe IP absorver o código do C-MAC e do ADHOP, que por sua vez são mais complexos que o código necessário para suportar Ethernet usando ARP. Vale a pena ressaltar que nem todo código gerado será transportado para o binário final da aplicação, isto se deve ao fato de aplicarmos técnicas de otimização do GCC durante a compilação e a link-edição, que fazem com que a imagem final das aplicações possua apenas as funções necessárias para seu funcionamento. Tais técnicas são conhecidas como *whole-program optmizations*.

Componente	Tamanho (bytes)
IP	12376
ICMP	3043
UDP	5331
TCP	13334
DHCP	1748

Tabela 1. Tamanho total dos arquivos-objeto de cada componente para arquitetura Intel x86 utilizando Ethernet e ARP.

A análise de algumas aplicações simples permite ter uma melhor ideia do espaço total ocupado, já que cada aplicação carrega apenas a funcionalidade necessária para desempenhar sua função. A tabela 3 mostra o tamanho total de três aplicativos de teste,

Componente	Tamanho (bytes)
IP	12224
ICMP	2364
UDP	4920
TCP	14364
DHCP	1692

Tabela 2. Tamanho total dos arquivos-objeto de cada componente para arquitetura ARM utilizando ADHOP e CMAC.

utilizando protocolos convencionais e interoperando com outros sistemas. O aplicativo Ping/Pong realiza requisições de ICMP ECHO e dá respostas ICMP ECHO-REPLY. O cliente DHCP faz auto-configuração de rede através do protocolo DHCP, que por sua vez foi implementado com UDP::Socket. A diferença entre o cliente WEB 1 e 2 é que o primeiro utiliza TCP::Socket e o segundo TCP::Channel.

Aplicação	Tamanho x86 (bytes)	Tamanho ARM (bytes)
Ping/Pong	27652	38260
Cliente DHCP	31044	41084
Cliente WEB 1	36368	47676
Cliente WEB 2	38768	49452

Tabela 3. Tamanho total dos arquivos-objeto de cada aplicativo de teste.

Classe	Memória RAM ocupada (bytes)
IP	184
ICMP	32
UDP	32
TCP	32
UDP::Socket	28
TCP::Socket	108
UDP::Channel	76
TCP::Socket	172

Tabela 4. Memória ocupada por cada instancia das classes da hierarquia de protocolos.

Na tabela 4 encontra-se o *footprint* de memória de cada objeto instanciado das classes da arquitetura TCP/IP. Como nossa arquitetura não impõe nenhum limite em tempo de compilação na quantidade máxima de conexões abertas, tal informação pode ser usada para o desenvolvedor impor algum limite em tempo de execução e planejar a capacidade do sistema.

5. Conclusão

Neste trabalho foi apresentada a pilha de protocolos TCP/IP desenvolvida para o sistema EPOS. Tal implementação é de grande utilidade para compatibilidade com sistemas

legados e interoperabilidade com sistemas diversos. Os resultados obtidos mostram a viabilidade de utilização na atual plataforma EPOSMote2, projetada para atender demandas acadêmicas e industriais nas mais diversas áreas de sistemas profundamente embarcados. Atualmente já há um trabalho em desenvolvimento utilizando esta pilha de protocolos como base para comunicação SIP/RTP e IEEE1451, o que ajudará a consolidar os artefatos desenvolvidos neste trabalho.

Por fim, podemos dizer que a ideia central do título de criar uma pilha flexível, configurável e de baixo custo para sistemas embarcados foi alcançada pelo desenvolvimento de uma elaborada hierarquia que permite ao desenvolvedor usar apenas o necessário para alcançar seus objetivos, ou seja, flexível e configurável. A utilização de algoritmos e estruturas de dados simples, resultando em pouco código, pouca utilização de CPU e RAM, confere o toque final de baixo custo ao trabalho.

5.1. Trabalhos futuros

Devida a limitação natural de tempo e recursos humanos de um TCC, foi necessário restringir o escopo do trabalho à apenas uma base conceitual e de software suficiente para garantir sua utilidade e permitir expansão futura. A continuidade natural deste trabalho será o desenvolvimento do suporte a IPv6 e a camada de adaptação 6lowPAN para integração de IPv6 em redes IEEE 802.15.4. A arquitetura desenvolvida foi propositalmente desacoplada para permitir a evolução da camada de rede de forma transparente à camada de transporte, diferentemente de outras implementações para sistemas embarcados e com a possibilidade de um crescimento incremental da complexidade do software envolvido.

Referências

ABDELJAOUAD, I. et al. Performance analysis of modern TCP variants: A comparison of Cubic, Compound and New Reno. *25th Biennial Symposium on Communications (QBSC)*, Kingston, ON, USA, p. 80–83, May 2010. ISSN 978-1-4244-5709-0.

DUNKELS, A. Full TCP/IP for 8 Bit Architectures. In: *Proceedings of the First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*. San Francisco: [s.n.], 2003. Disponível em: <http://www.sics.se/ adam/mobisys2003.pdf>.

DUNKELS, A. *Contiki: Bringing IP to Sensor Networks*. jan. 2009. ERCIM News. Disponível em: <http://ercim-news.ercim.org/content/view/496/705/>.

DUNKELS, A.; GRÖNVALL, B.; VOIGT, T. Contiki - a lightweight and flexible operating system for tiny networked sensors. In: *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*. Tampa, Florida, USA: [s.n.], 2004. Disponível em: <http://www.sics.se/ adam/dunkels04contiki.pdf>.

DUNKELS, A.; SCHMIDT, O. *Protothreads - Lightweight Stackless Threads in C*. [S.l.], mar. 2005. Disponível em: <http://www.sics.se/ adam/dunkels05protothreads.pdf>.

DUNKELS, A. et al. Connecting Wireless Sensornets with TCP/IP Networks. In: *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*. Frankfurt (Oder), Germany: [s.n.], 2004. (C) Copyright 2004 Springer Verlag. http://www.springer.de/comp/lncs/index.html. Disponível em: <http://www.sics.se/ adam/wwic2004.pdf>.

EUGSTER, P. T. et al. The many faces of publish/subscribe. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 35, p. 114–131, June 2003. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/857076.857078>.

FRÖHLICH, A. A. *Application-Oriented Operating Systems*. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, 2001. 200 p. ISBN 3-88457-400-0.

GONT, F.; YOURTCHENKO, A. *On the Implementation of the TCP Urgent Mechanism*. IETF, jan. 2011. RFC 6093 (Proposed Standard). (Request for Comments, 6093). Disponível em: <http://www.ietf.org/rfc/rfc6093.txt>.

GOODE, B. Voice over internet protocol (voip). *Proceedings of the IEEE*, IEEE, v. 90, n. 9, p. 1495–1517, 2002. Disponível em: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1041060>.

HA, S.; RHEE, I.; XU, L. Cubic: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 42, p. 64–74, July 2008. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/1400097.1400105>.

HOWES, T.; SMITH, M. *The LDAP Application Program Interface*. IETF, ago. 1995. RFC 1823 (Informational). (Request for Comments, 1823). Disponível em: <http://www.ietf.org/rfc/rfc1823.txt>.

KENT, S.; ATKINSON, R. *Security Architecture for the Internet Protocol*. IETF, nov. 1998. RFC 2401 (Proposed Standard). (Request for Comments, 2401). Obsoleted by RFC 4301, updated by RFC 3168. Disponível em: <http://www.ietf.org/rfc/rfc2401.txt>.

LI, Y.-C.; CHIANG, M.-L. Lyranet: A zero-copy tcp/ip protocol stack for embedded operating systems. *Real-Time Computing Systems and Applications, International Workshop on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 123–128, 2005. ISSN 1533-2306.

MASCOLO, S. Modeling the internet congestion control using a smith controller with input shaping. *Control Engineering Practice*, v. 14, n. 4, p. 425 – 435, 2006. ISSN 0967-0661. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0967066105000572>.

MOCKAPETRIS, P. *Domain names - implementation and specification*. IETF, nov. 1987. RFC 1035 (Standard). (Request for Comments, 1035). Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966. Disponível em: <http://www.ietf.org/rfc/rfc1035.txt>.

OKAZAKI, A. M.; FRÖHLICH, A. A. Adapting HOPNET Algorithm for Wireless Sensor Networks. In: *International Information and Telecommunication Technologies Symposium*. Florianópolis, Brazil: [s.n.], 2009. p. 191–194. ISBN 978-85-89264-10-5.

OKAZAKI, A. M.; FRÖHLICH, A. A. AD-ZRP: Ant-based Routing Algorithm for Dynamic Wireless Sensor Networks. In: *18th International Conference on Telecommunications*. Ayia Napa, Cyprus: [s.n.], 2011. p. 15–20. ISBN 978-1-4577-0023-1.

ONG, L.; YOAKUM, J. *An Introduction to the Stream Control Transmission Protocol (SCTP)*. IETF, maio 2002. RFC 3286 (Informational). (Request for Comments, 3286). Disponível em: <http://www.ietf.org/rfc/rfc3286.txt>.

POLPETA, F. V.; FRÖHLICH, A. A. Hardware Mediators: a Portability Artifact for Component-Based Systems. In: *International Conference on Embedded and Ubiquitous Computing*. Aizu, Japan: Springer, 2004. (Lecture Notes in Computer Science, v. 3207), p. 271–280. ISBN 354022906x.

POSTEL, J. *User Datagram Protocol*. IETF, ago. 1980. RFC 768 (Standard). (Request for Comments, 768). Disponível em: <http://www.ietf.org/rfc/rfc768.txt>.

POSTEL, J. *Internet Protocol*. IETF, set. 1981. RFC 791 (Standard). (Request for Comments, 791). Updated by RFC 1349. Disponível em: <http://www.ietf.org/rfc/rfc791.txt>.

POSTEL, J. *Transmission Control Protocol*. IETF, set. 1981. RFC 793 (Standard). (Request for Comments, 793). Updated by RFCs 1122, 3168, 6093. Disponível em: <http://www.ietf.org/rfc/rfc793.txt>.

SHELBY, Z. et al. NanoIP: The Zen of Embedded Networking. *PROCEEDINGS OF THE IEEE*, p. 1218–1222, 2003.

TAN, K.; SONG, J. Compound TCP: A scalable and TCP-friendly congestion control for high-speed networks. In: *in 4th International workshop on Protocols for Fast Long-Distance Networks (PFLDNet), 2006*. [S.l.: s.n.], 2006.

TANENBAUM, A. *Computer Networks*. 4th. ed. [S.l.]: Prentice Hall Professional Technical Reference, 2002. ISBN 0130661023.

VASSEUR, J.-P.; DUNKELS, A. *Interconnecting Smart Objects with IP - The Next Internet*. Morgan Kaufmann, 2010. ISBN 978-0123751652. Disponível em: <http://TheNextInternet.org/>.