Aula 8 - Conceitos de Camada de Transporte, UDP, Transferência Confiável (I)

Diego Passos

Universidade Federal Fluminense

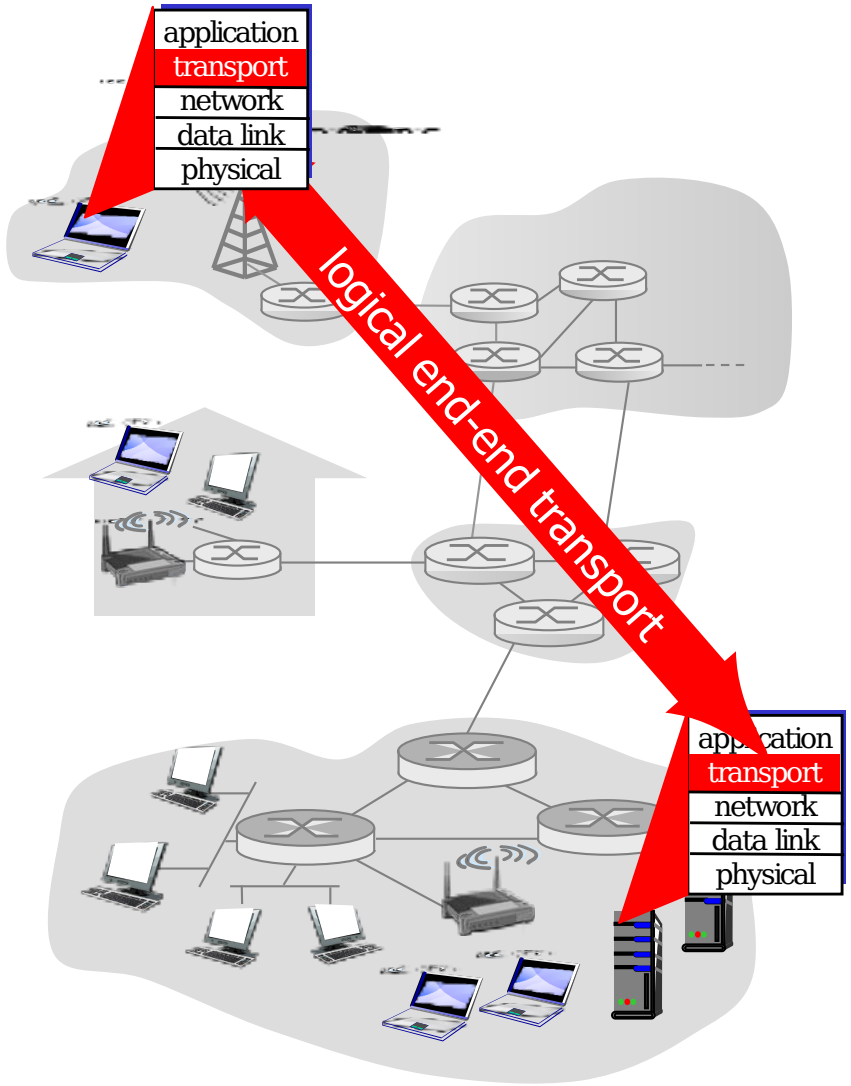
Redes de Computadores

Material adaptado a partir dos slides originais de J.F Kurose and K.W. Ross.

Serviços da Camada de Transporte

Serviços e Protocolos de Transporte

- Provê **comunicação lógica** entre **processos** da aplicação rodando em *hosts* diferentes.
- Protocolos de transporte **são executados nos sistemas finais**.
  - Lado transmissor: quebra mensagens da aplicação em **segmentos**, passa segmentos para camada de rede.
  - Lado receptor: remonta segmentos para formar mensagens originais, passa mensagens para a camada de aplicação.
- Mais de um protocolo disponível para as aplicações.
  - Na Internet: TCP e UDP.



Camada de Transporte *vs.* Camada de Rede

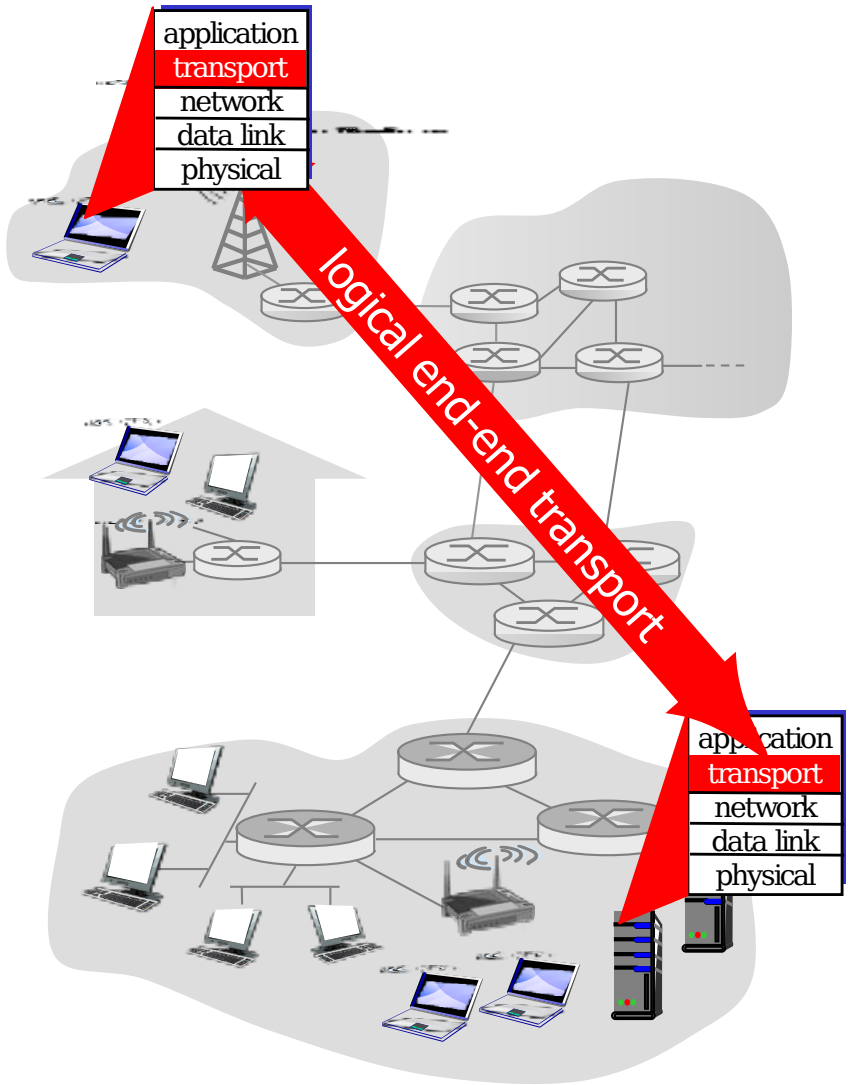
- **Camada de Rede:** comunicação lógica entre *hosts*.
- **Camada de Transporte:** comunicação lógica entre processos.
  - Depende de, e aperfeiçoa, serviços da camada de rede.

Analogia doméstica

- 12 crianças na casa da Ann enviam cartas a 12 crianças na casa do Bill.
  - *hosts* = casas.
  - processos = crianças.
  - mensagens da aplicação = cartas nos envelopes.
  - protocolo de transporte = Ann e Bill que demultiplexam cartas para as crianças.
  - protocolo de camada de rede = correios.

Protocolos de Camada de Transporte da Internet

- Entrega confiável, em ordem (TCP).
  - Controle de congestionamento.
  - Controle de fluxo.
  - *Setup* da conexão.
- Entrega não-confiável, não-ordenada (UDP).
  - Extensão básica do serviço de "melhor esforço" do IP.
- Serviços não disponíveis (nem TCP, nem UDP):
  - Garantias de atraso máximo.
  - Garantias de vazão mínima.



Multiplexação e Demultiplexação

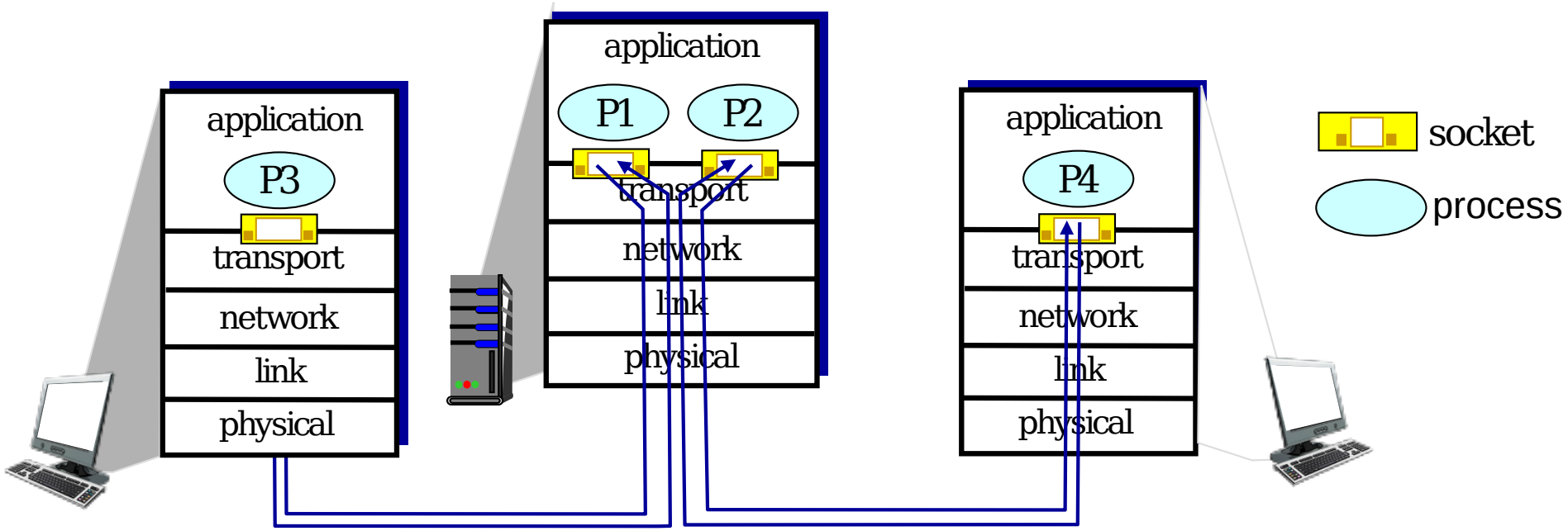
Multiplexação/Demultiplexação

Multiplexação no Transmissor

Lida com dados de múltiplos sockets, adiciona cabeçalho da camada de transporte (usado posteriormente para demultiplexação)

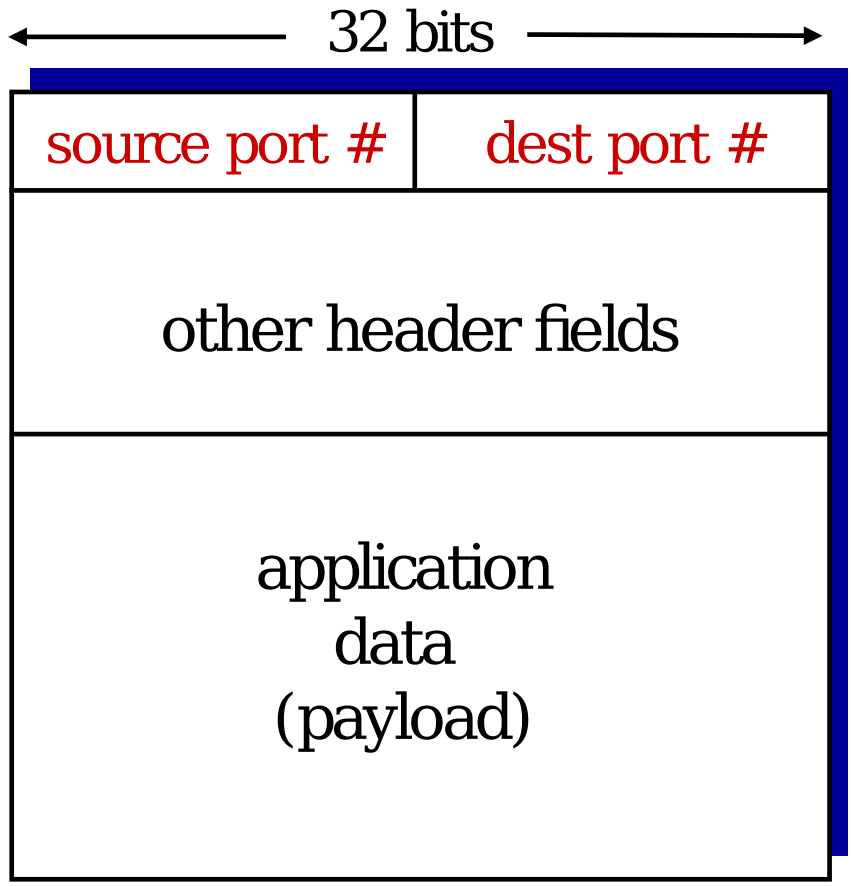
Demultiplexação no Receptor

Usa informação do cabeçalho para entregar segmentos recebidos para o socket correto



Como a Demultiplexação Ocorre

- *Host* recebe datagrama IP.
  - Cada datagrama possui um endereço IP de origem, endereço IP de destino.
  - Cada datagrama carrega um segmento de camada de transporte.
  - Cada segmento possui **números de porta de origem e de destino**.
- *Host* utiliza **tanto os endereços IP quanto os números de porta** para direcionar segmentos aos sockets apropriados.



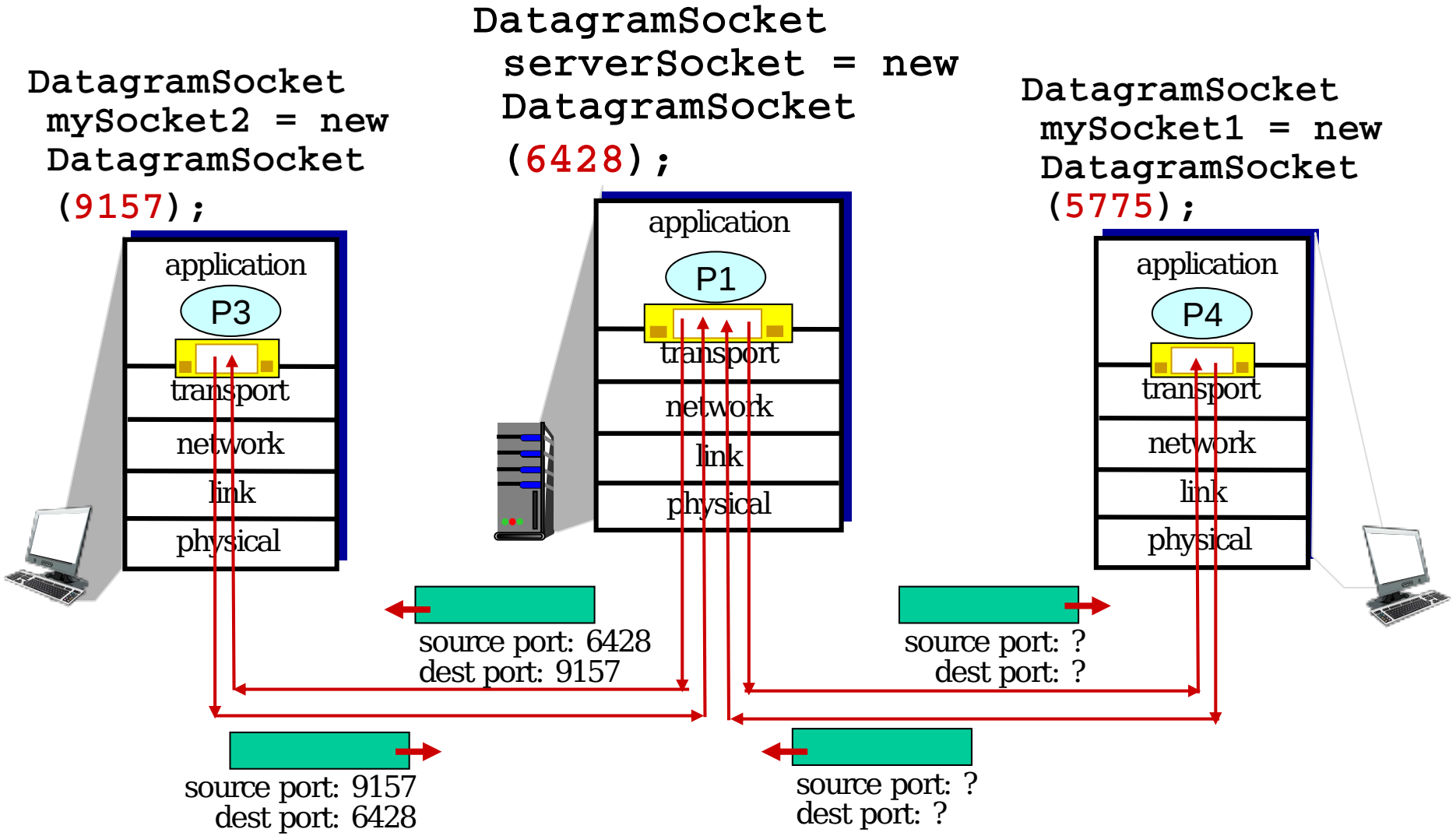
TCP/UDP segment format

Demultiplexação Sem Conexão

- **Lembre-se:** socket criado tem # de porta no *host*.
  `DatagramSocket mySocket1 = new DatagramSocket(12534);`
- **Lembre-se:** quando criamos datagrama para enviar pelo socket UDP, é preciso especificar:
  - Endereço IP de destino.
  - # de porta de destino.

- Quando *host* recebe segmento UDP:
  - Verifica o # de porta de destino no segmento.
  - Direciona o segmento UDP para o socket com aquele # de porta.
- Datagramas com **o mesmo # de porta de destino**, mas com IPs e/ou portas de origem diferentes serão direcionados **ao mesmo socket** no destinatário.

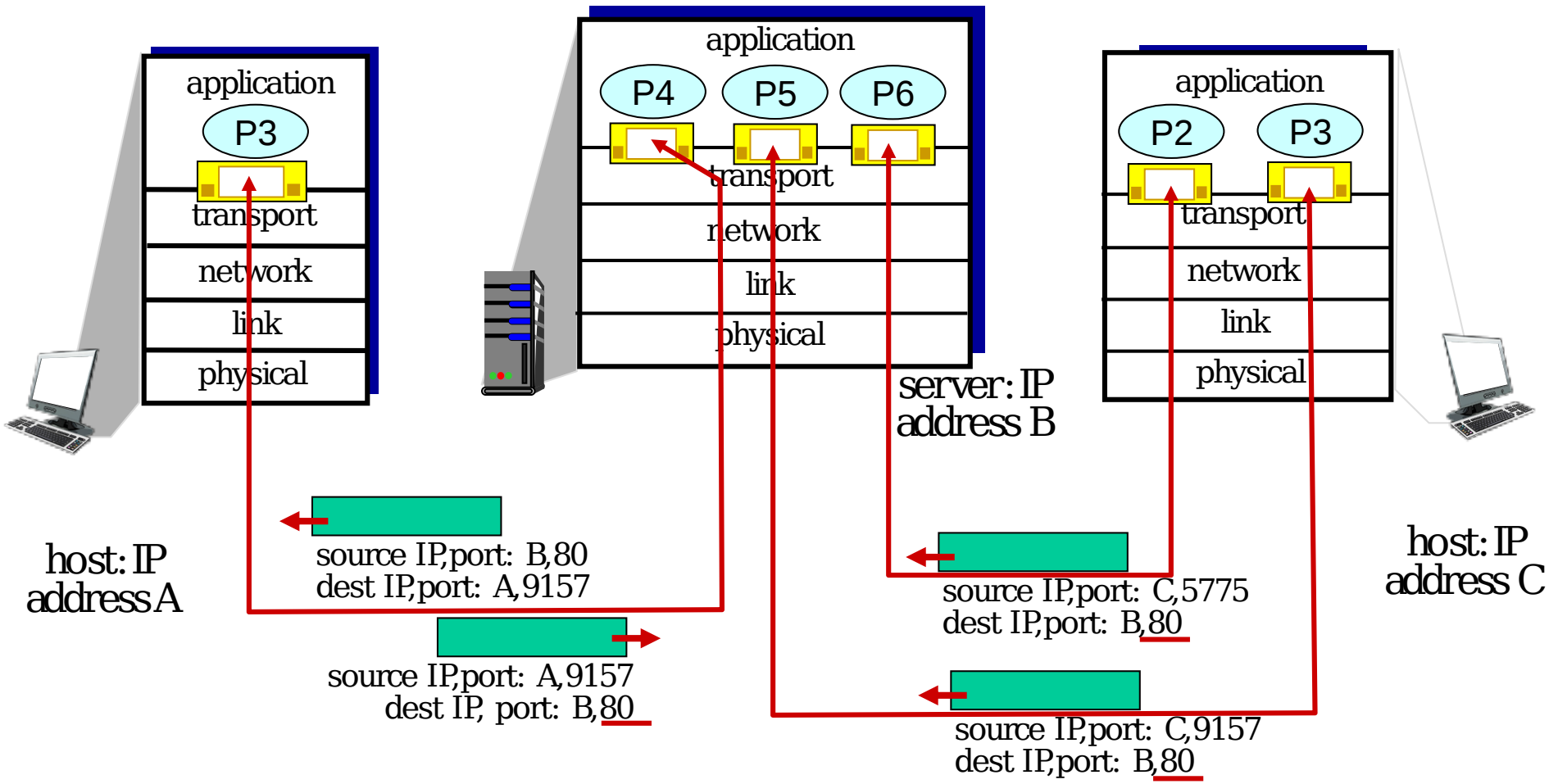
Demultiplexação Sem Conexão: Exemplo



Demultiplexação Orientada a Conexão

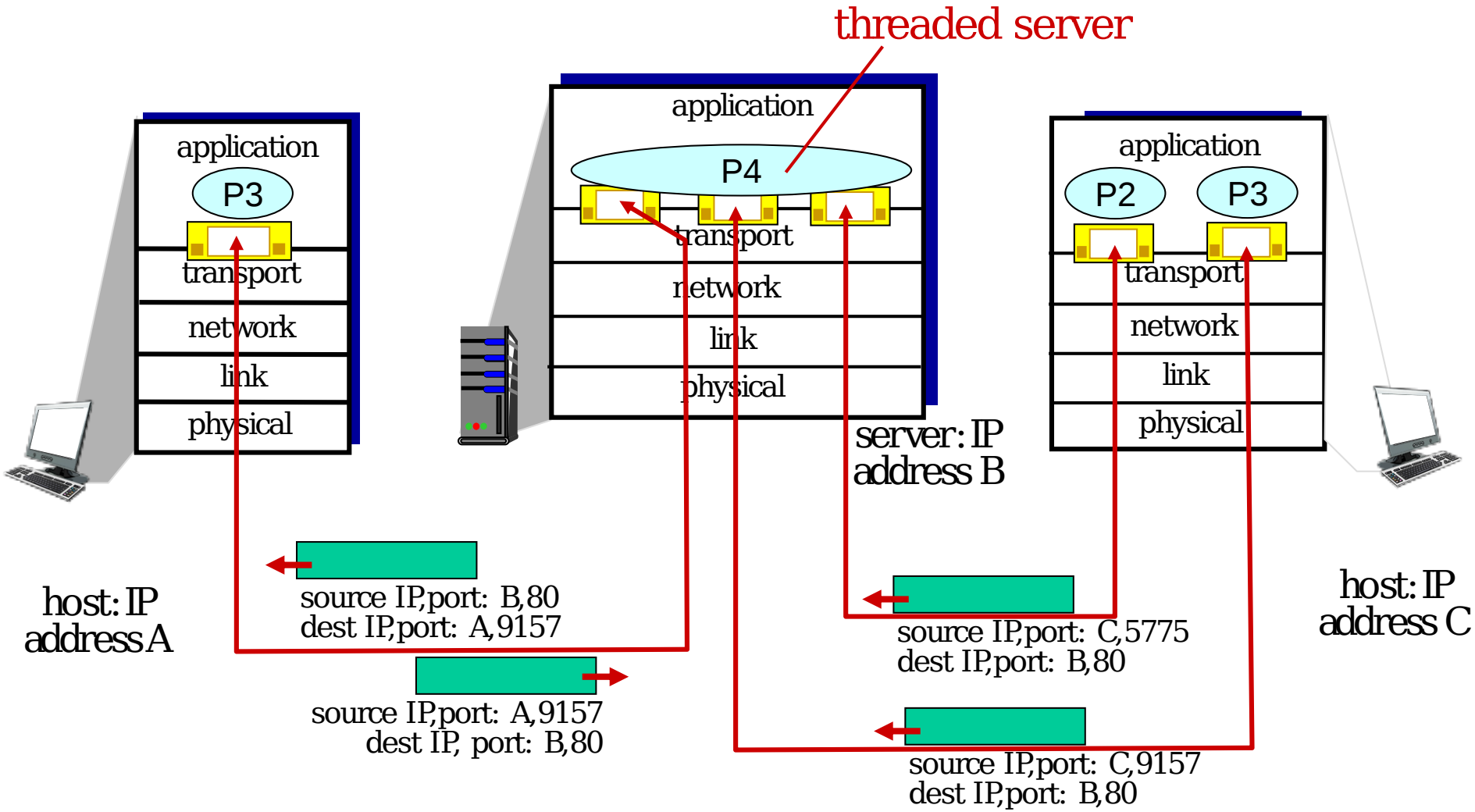
- Socket TCP é identificado por tupla de 4 componentes:
  - **IP de origem**.
  - **IP de destino**.
  - **Porta de origem**.
  - **Porta de destino**.
- Demultiplexação: receptor usa todos os quatro valores para direcionar segmento a socket correto.
- *Host* servidor pode suportar múltiplos sockets TCP simultâneos.
  - Cada socket identificado pela sua própria tupla de quatro valores.
- Servidores web têm sockets diferentes para cada cliente conectado.
  - No HTTP não-persistente, um socket para cada requisição.

Demultiplexação Orientada a Conexão: Exemplo



Três segmentos, todos destinados ao IP de B na porta de destino 80 são demultiplexados para sockets diferentes.

Demultiplexação Orientada a Conexão: Exemplo (Threads)

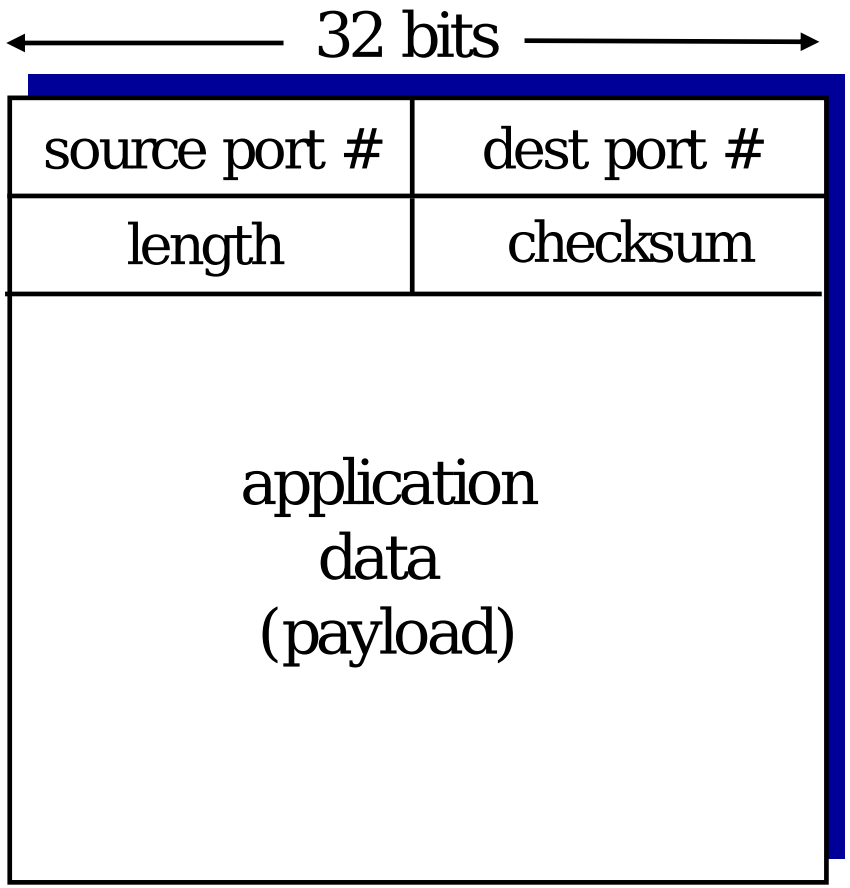


Transporte Sem Conexão: UDP

UDP: *User Datagram Protocol* [RFC 768]

- "Serviço básico", "mínimo" da camada de transporte da Internet.
- Modelo de serviço de "melhor esforço".
- Segmentos UDP podem ser:
  - Perdidos
  - Entregues, porém fora de ordem para a aplicação.
- **Sem conexão:**
  - Não há comunicação inicial entre UDP do transmissor e do receptor.
  - Datagramas são simplesmente enviados.
  - Cada segmento UDP é tratado de forma completamente independente dos demais.
- Usos do UDP:
  - Aplicações de *Streaming* multimídia (tolerantes a perda, sensíveis a taxa).
  - DNS.
  - SNMP.
- Transferência confiável sobre UDP:
  - Possível, mas depende da aplicação.
  - Adição de confiabilidade da própria aplicação.
  - Métodos de recuperação de erros específicos de cada aplicação.

UDP: Cabeçalho de um Segmento



UDP segment format

- **Campo *length***: tamanho do segmento, incluindo cabeçalhos.

Por que existe um UDP?

- Sem estabelecimento de conexão (que adiciona atraso).
- Simples: não armazena estado da comunicação no transmissor ou no receptor.
- Cabeçalho pequeno.
- Sem controle de congestionamento: UDP transmite na mesma taxa que a aplicação gera.

UDP: *Checksum*

- **Objetivo:** detectar "erros" (*e.g.*, bits com valor trocado) no segmento transmitido.

- **Transmissor:**
  - Trata conteúdo do segmento, incluindo campos de cabeçalho, como uma sequência de inteiros de 16 bits.
  - *Checksum*: soma, em complemento a 1, do conteúdo do segmento.
  - Transmissor coloca valor do *checksum* no campo do cabeçalho UDP.

- **Receptor:**
  - Computa o *checksum* do segmento recebido.
  - *Checksum* computado é igual ao indicado pelo cabeçalho?
    - Não: erro detectado.
    - Sim: nenhum erro detectado.
      - Mas pode haver erros mesmo assim? Mais detalhes em Redes II.

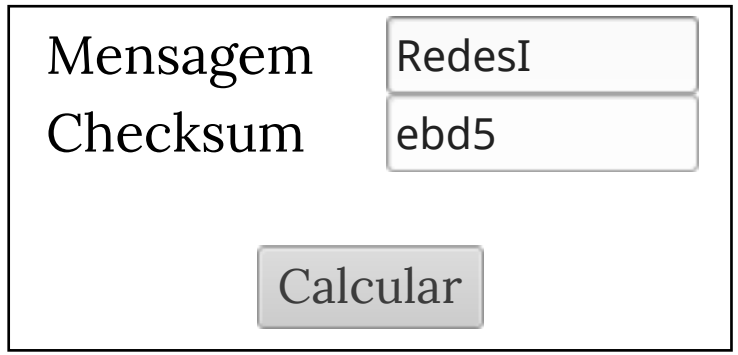
Checksum da Internet: Soma em Complemento a 1

- Soma de dois valores de 16 bits em complemento a 1:

- Note: ao somar dois números, o *vai-um* do bit mais significativo deve ser somado ao resultado.

Internet Checksum: Exemplos

- Experimente o cálculo do *checksum* de algumas mensagens (*strings*):

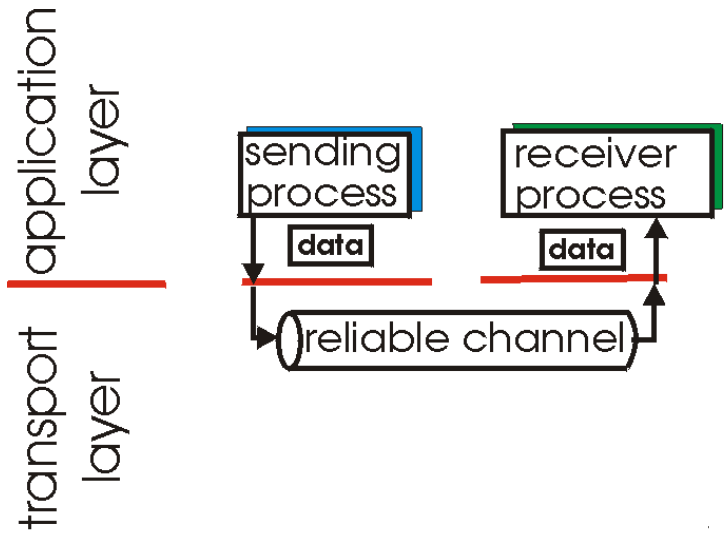


- Sugestão: calcule o *checksum* de “casa”.
  - Resultado: 0x3d29.
  - Em ASCII: 0x3D → “=”.
  - Em ASCII: 0x29 → “)”.
- **Pergunta:** qual é o *checksum* de “casa)=”?

Princípios de Transferência Confiável de Dados

Princípios de Transferência Confiável de Dados

- Importante nas camadas de aplicação, transporte e enlace.
  - Um dos 10 problemas mais importantes em redes de computadores!

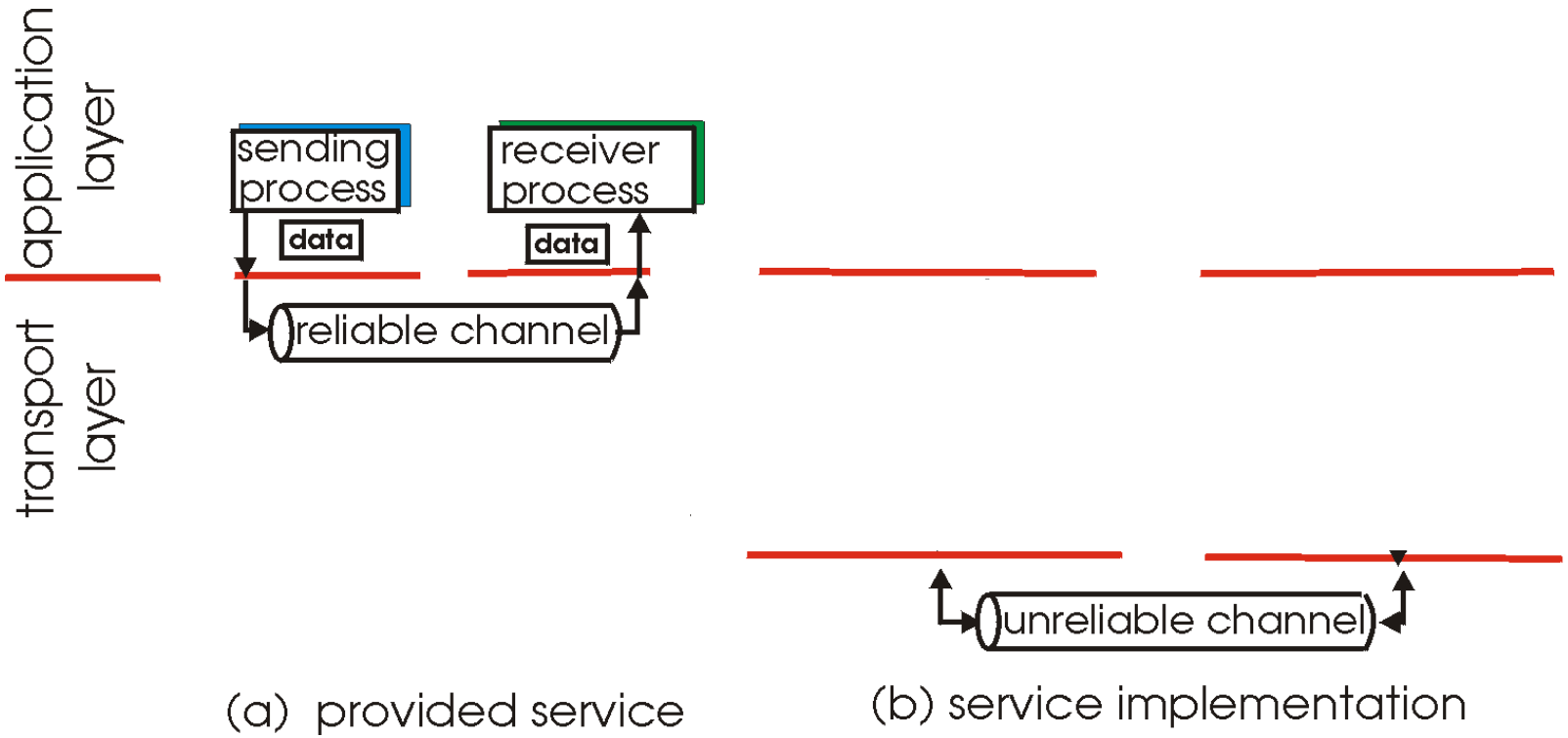


(a) provided service

- Características do canal não-confiável determinarão complexidade do protocolo de transmissão confiável de dados.
  - Ou rdt, do inglês *reliable data transfer*.

Princípios de Transferência Confiável de Dados

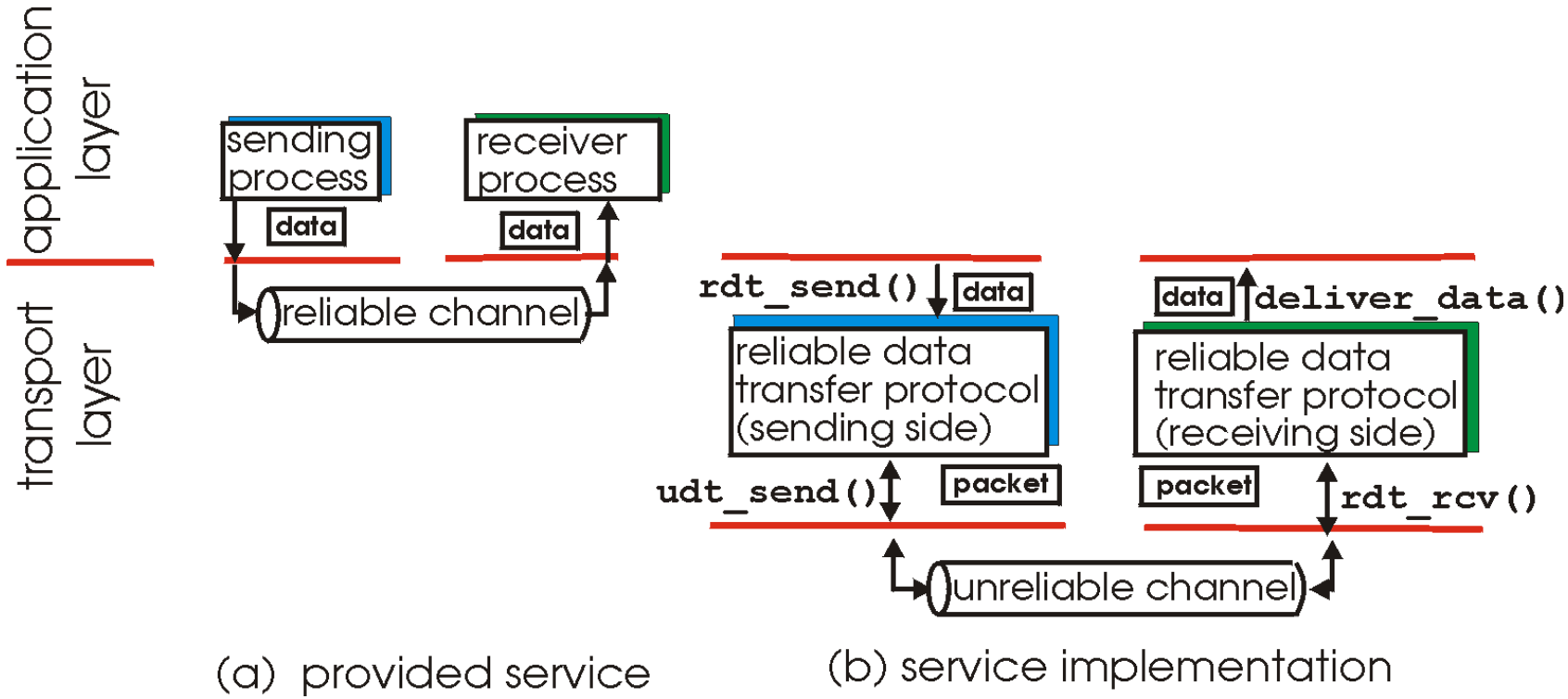
- Importante nas camadas de aplicação, transporte e enlace.
  - Um dos 10 problemas mais importantes em redes de computadores!



- Características do canal não-confiável determinarão complexidade do protocolo de transmissão confiável de dados.
  - Ou rdt, do inglês *reliable data transfer*.

Princípios de Transferência Confiável de Dados

- Importante nas camadas de aplicação, transporte e enlace.
  - Um dos 10 problemas mais importantes em redes de computadores!

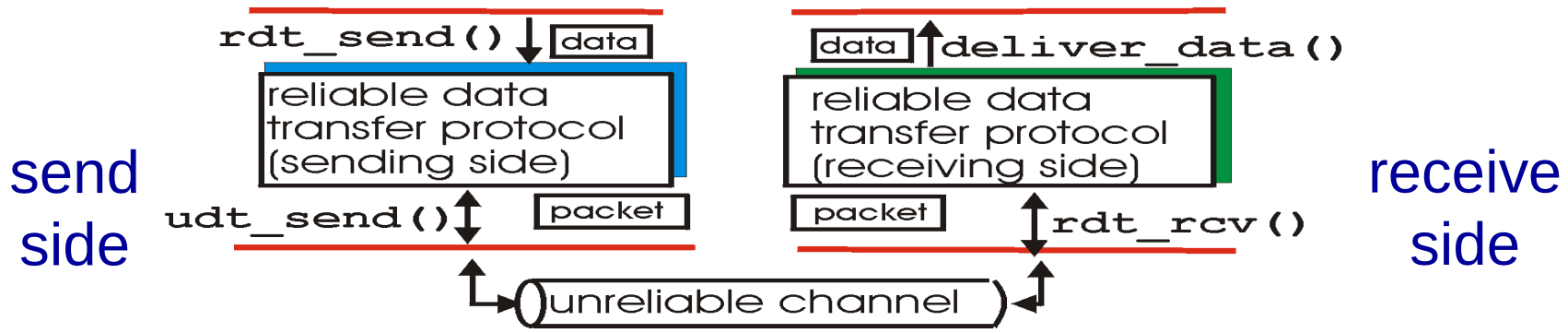


(a) provided service

(b) service implementation

- Características do canal não-confiável determinarão complexidade do protocolo de transmissão confiável de dados.
  - Ou rdt, do inglês *reliable data transfer*.

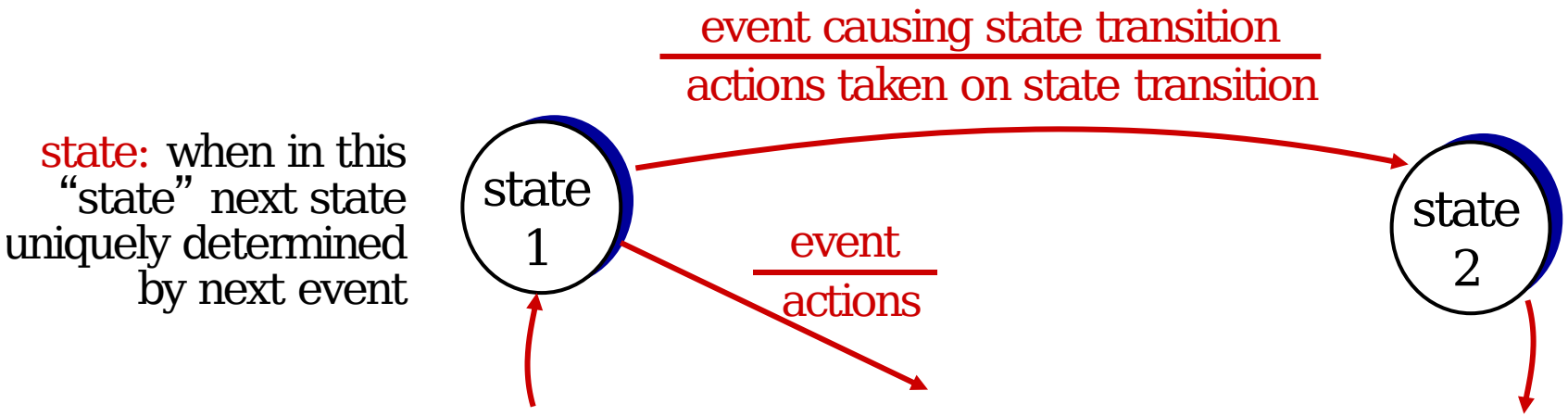
Transmissão Confiável de Dados: Início (I)



- **rdt_send():** chamada pela aplicação para enviar dados para o transporte.
- **udt_send():** chamado pelo transporte para passa pacote para a rede.
- **rdt_rcv():** chamada quando pacote chega pela rede no lado receptor.
- **deliver_data():** chamado pelo transporte para entregar dados para aplicação.

Transmissão Confiável de Dados: Início (II)

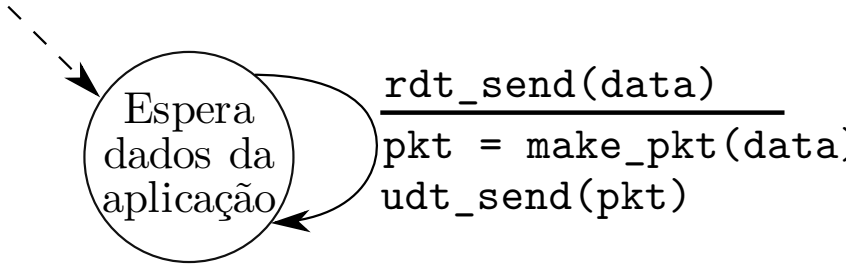
- Nós iremos:
  - Incrementalmente desenvolver os lados transmissor e receptor de um protocolo rdt.
  - Consideraremos apenas transmissão unidirecional de dados.
    - Mas informação de controle trafegará nos dois sentidos!
  - Usar máquinas de estado para especificar transmissor, receptor.



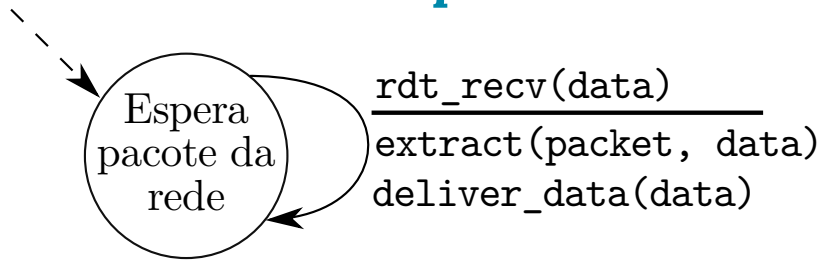
rdt1.0: Transmissão Confiável sobre Canal Confiável

- Canal de comunicação (rede) perfeitamente confiável.
  - Pacotes nunca são perdidos.
  - Sempre são entregues íntegros.
- Máquinas de estado separadas para transmissor, receptor:
  - Transmissor envia dados pelo canal.
  - Receptor lê dados a partir do canal.

Transmissor



Receptor



rdt2.0: Canal com Erros de Bit (I)

- Canal (rede) pode alterar valor de determinados bits.
  - Mas pacotes **sempre** são entregues, ainda que **corrompidos**.
- Já vimos uma maneira de verificar erros: *checksum*.
- Mas a pergunta é: como o protocolo **se recupera dos erros**?

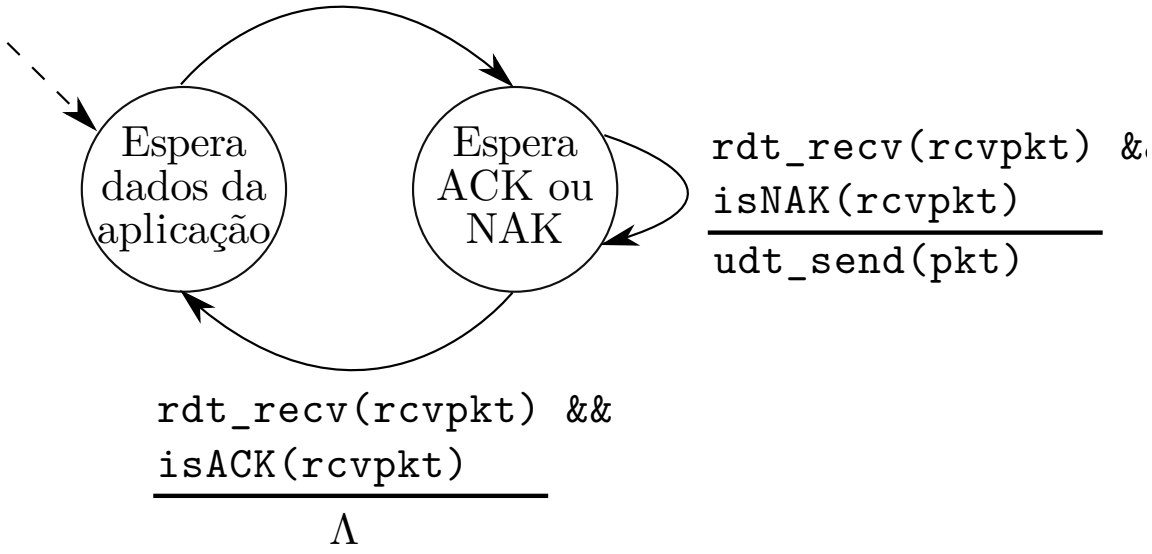
Como humanos se recuperam de "erros" durante uma conversa?

rdt2.0: Canal com Erros de Bit (II)

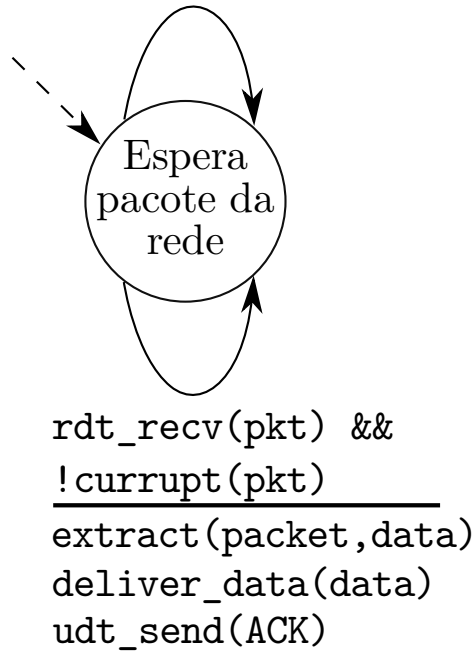
- Canal (rede) pode alterar valor de determinados bits.
  - Mas pacotes **sempre** são entregues, ainda que **corrompidos**.
- Já vimos uma maneira de verificar erros: *checksum*.
- Mas a pergunta é: como o protocolo **se recupera dos erros**?
  - **Pacotes de reconhecimento (ACKs):** receptor diz explicitamente ao transmissor que pacote foi recebido corretamente.
  - **Reconhecimento negativo (NAKs):** receptor diz explicitamente ao transmissor que pacote foi recebido com erros.
  - Transmissor retransmite pacote sempre que receber um NAK.
- Novo mecanismo no rdt2.0 (e versões posteriores):
  - Detecção de erros (via *checksum*).
  - Retro-alimentação: mensagens de controle (ACK, NAK) do receptor para o transmissor.

rdt2.0: Especificação da Máquina de Estados

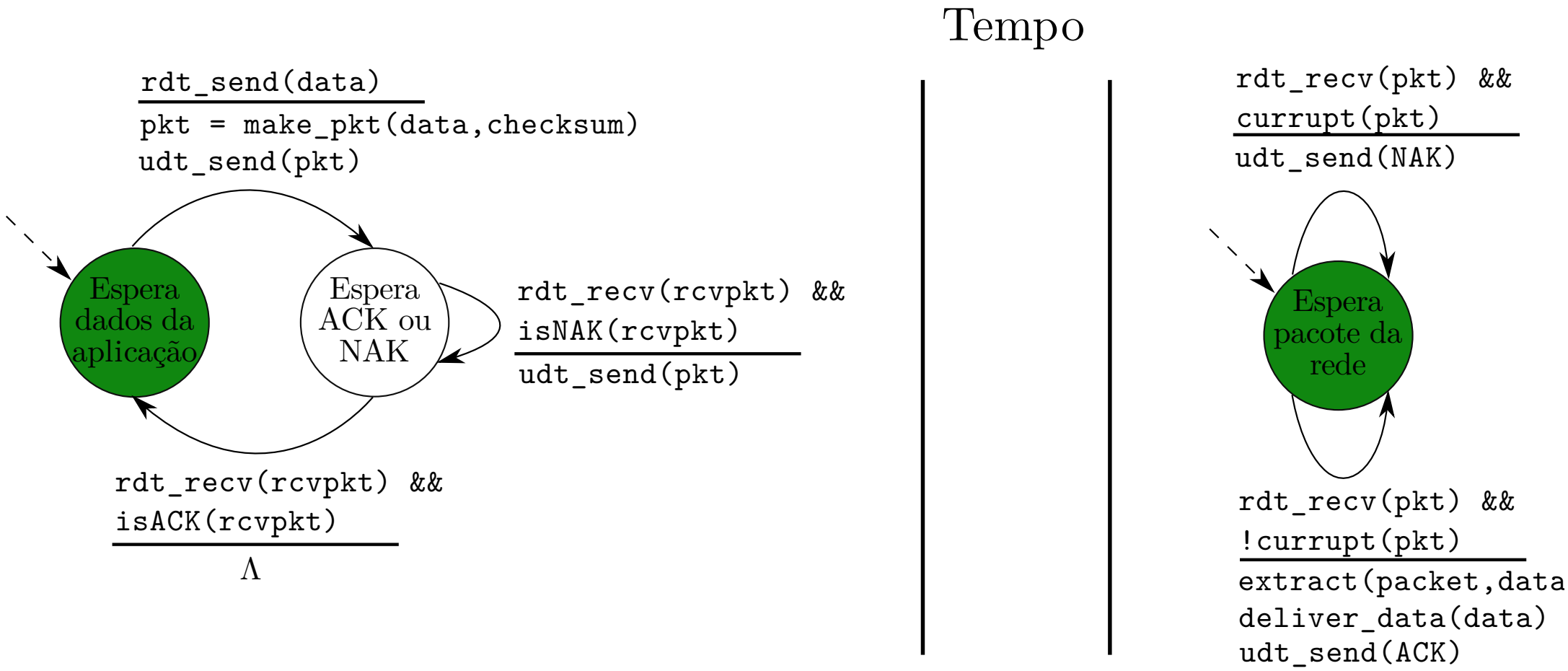
Transmissor



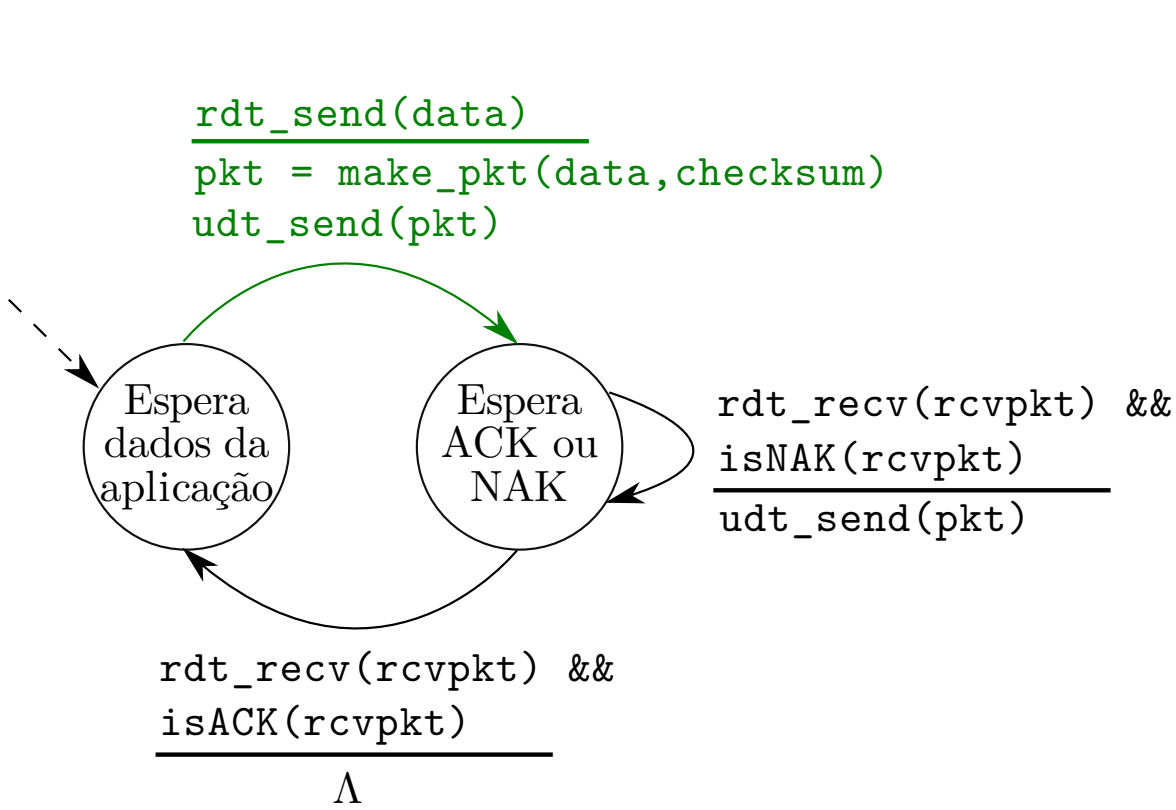
Receptor

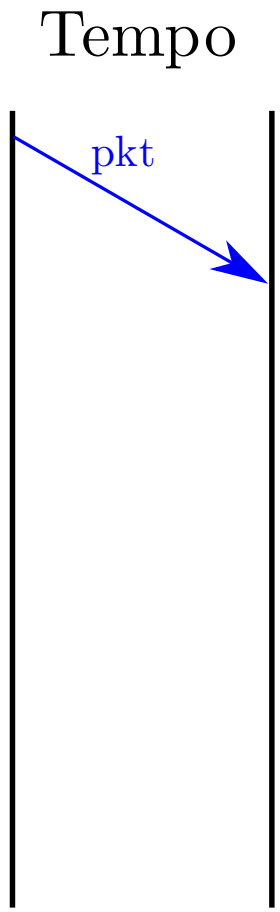


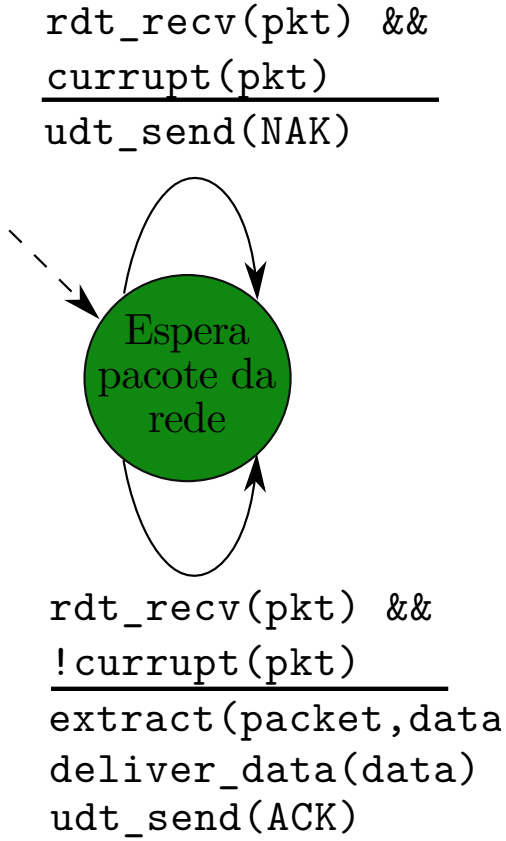
rdt2.0: Operação Sem Erros (I)



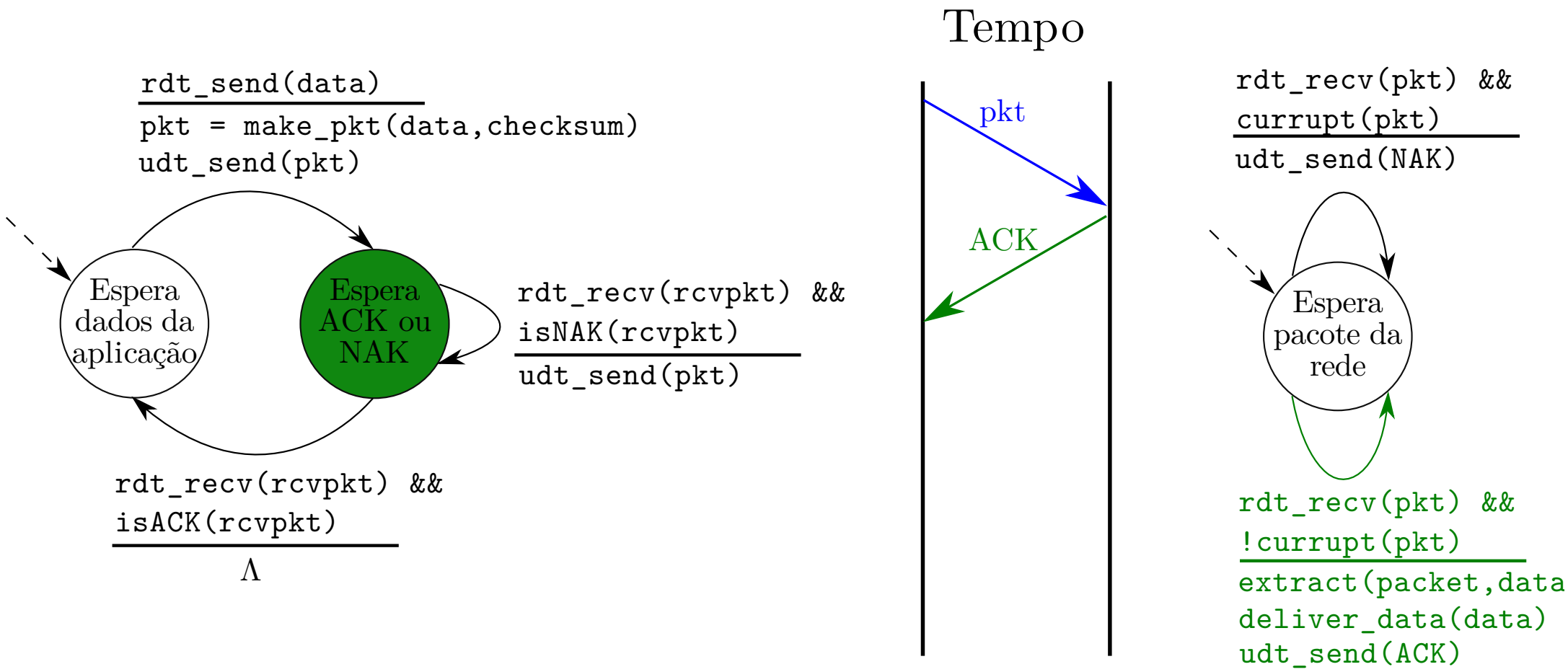
rdt2.0: Operação Sem Erros (II)



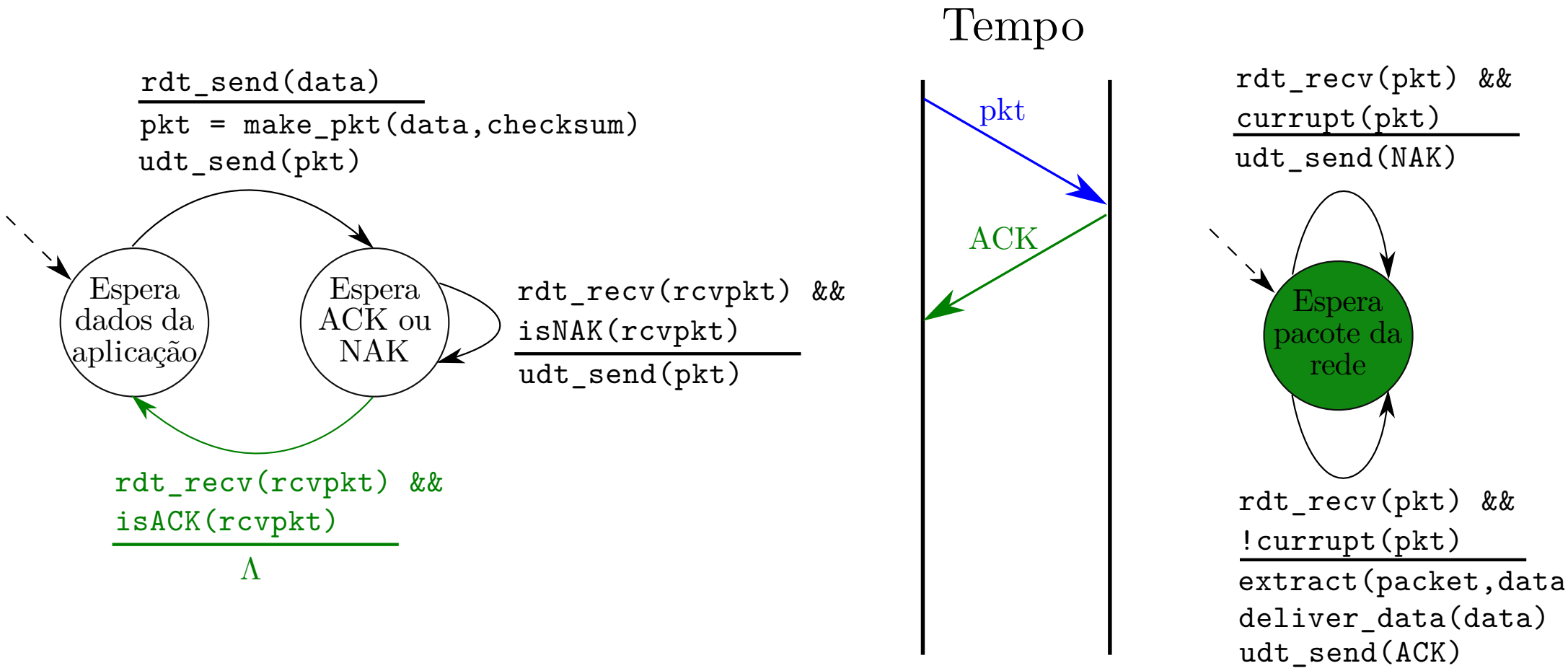




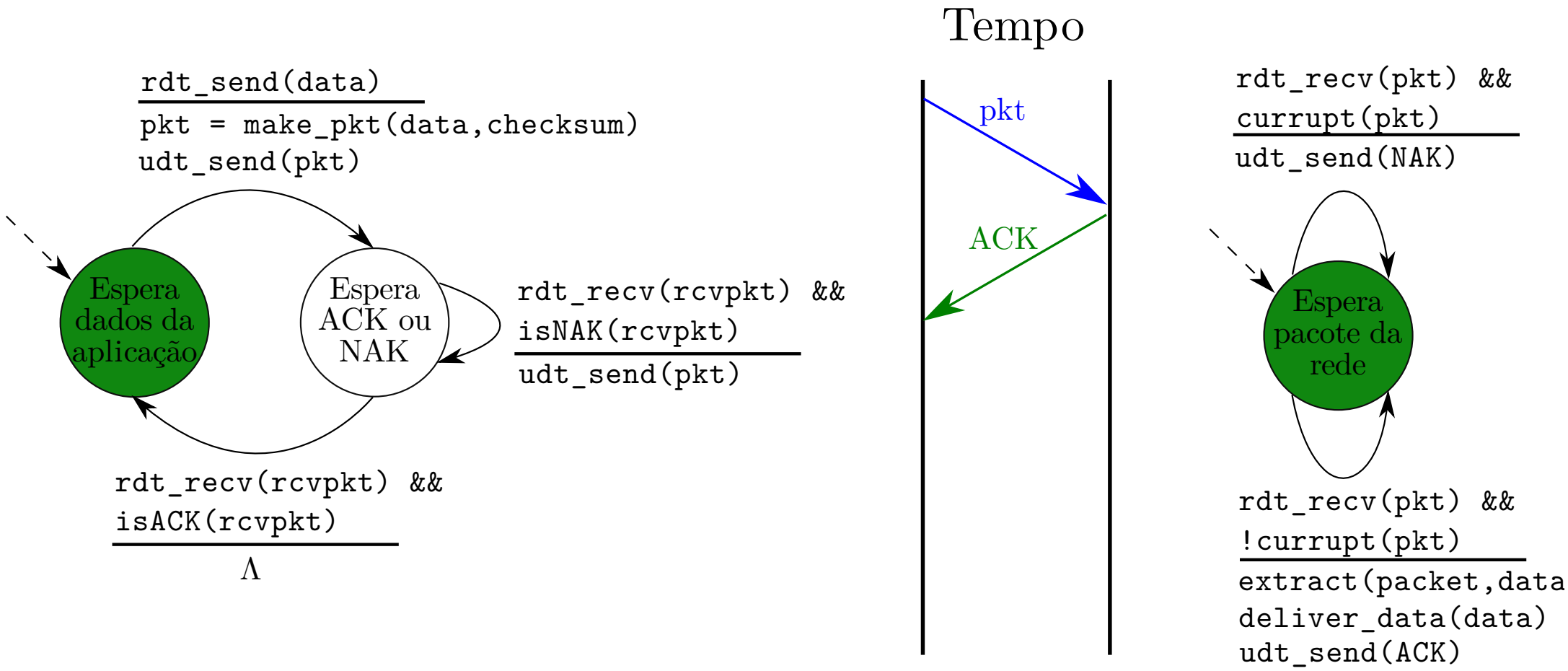
rdt2.0: Operação Sem Erros (III)



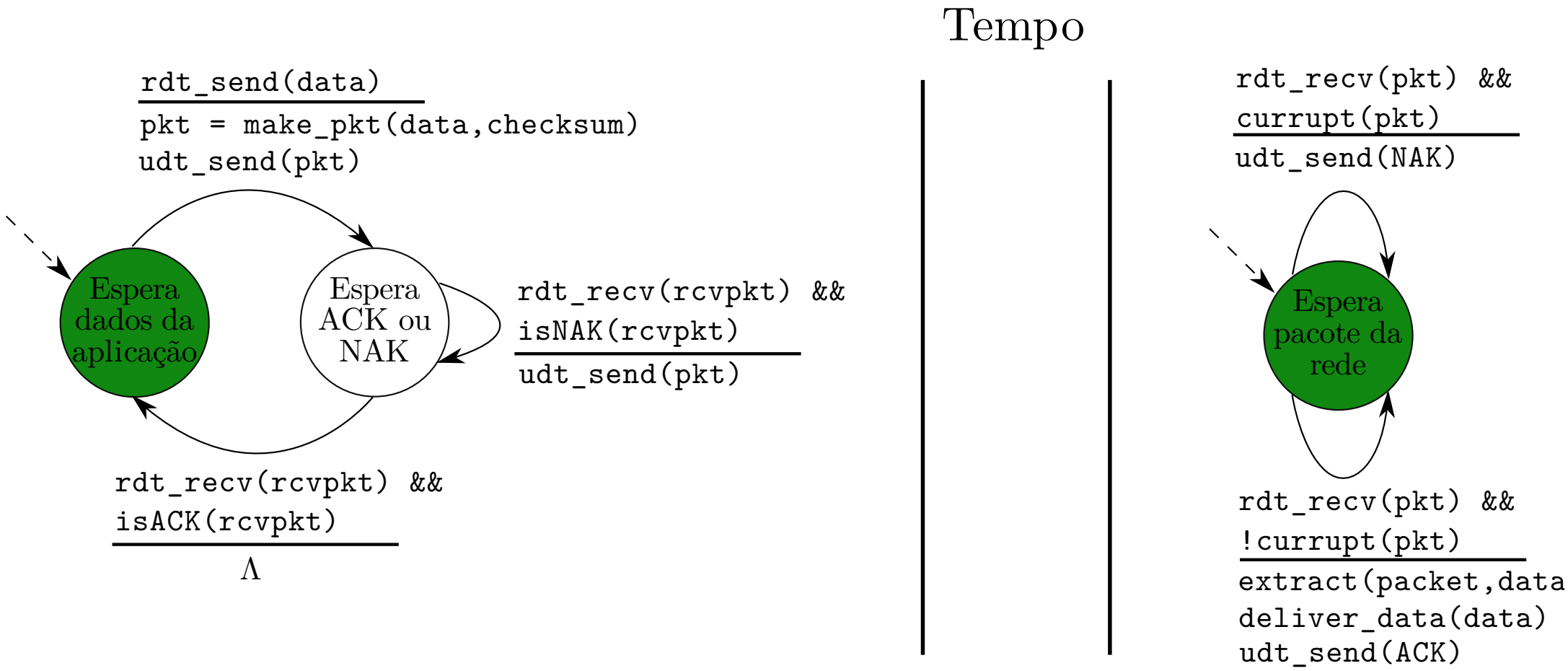
rdt2.0: Operação Sem Erros (IV)



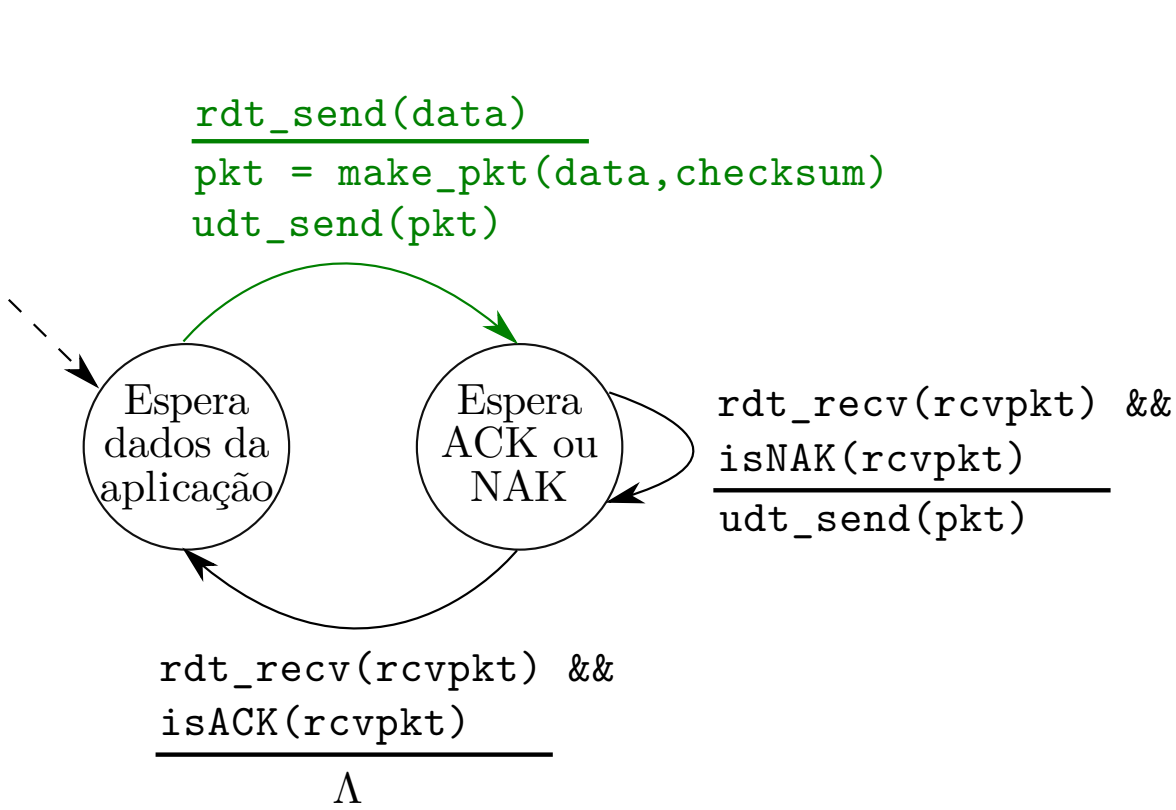
rdt2.0: Operação Sem Erros (V)

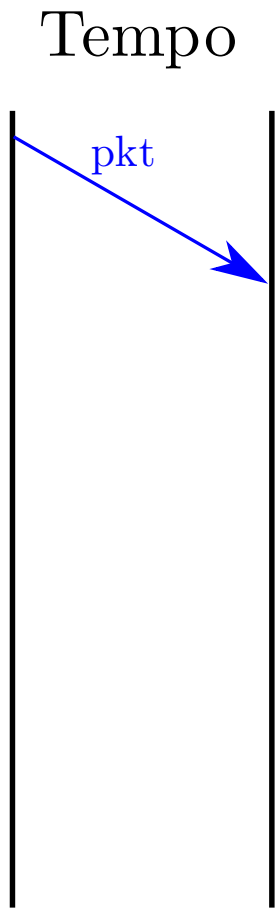


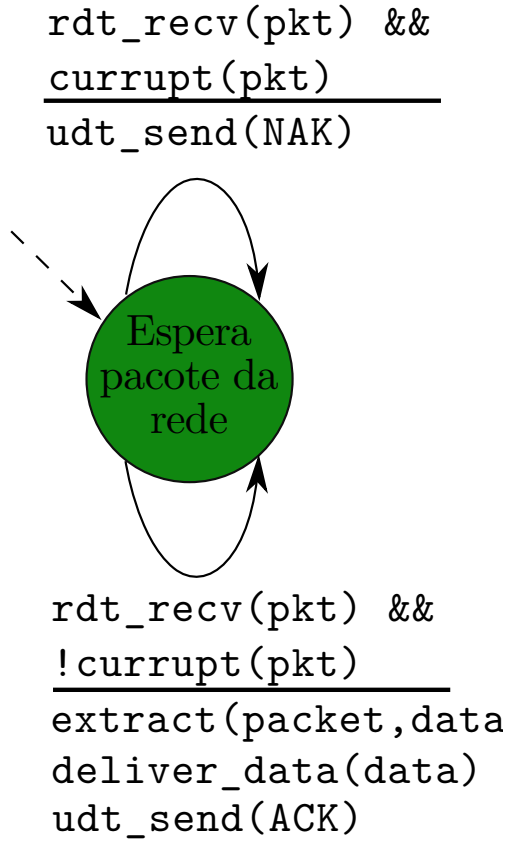
rdt2.0: Operação Com Erros (I)



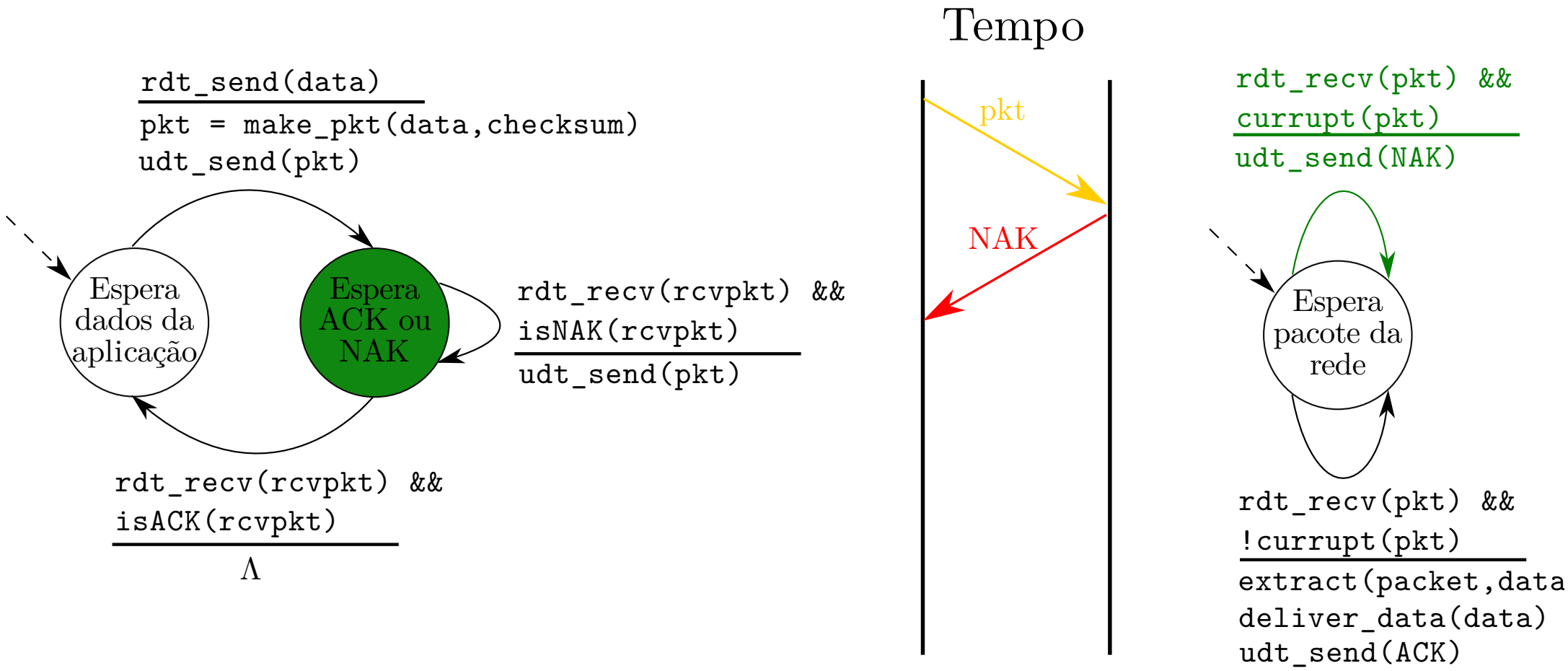
rdt2.0: Operação Com Erros (II)



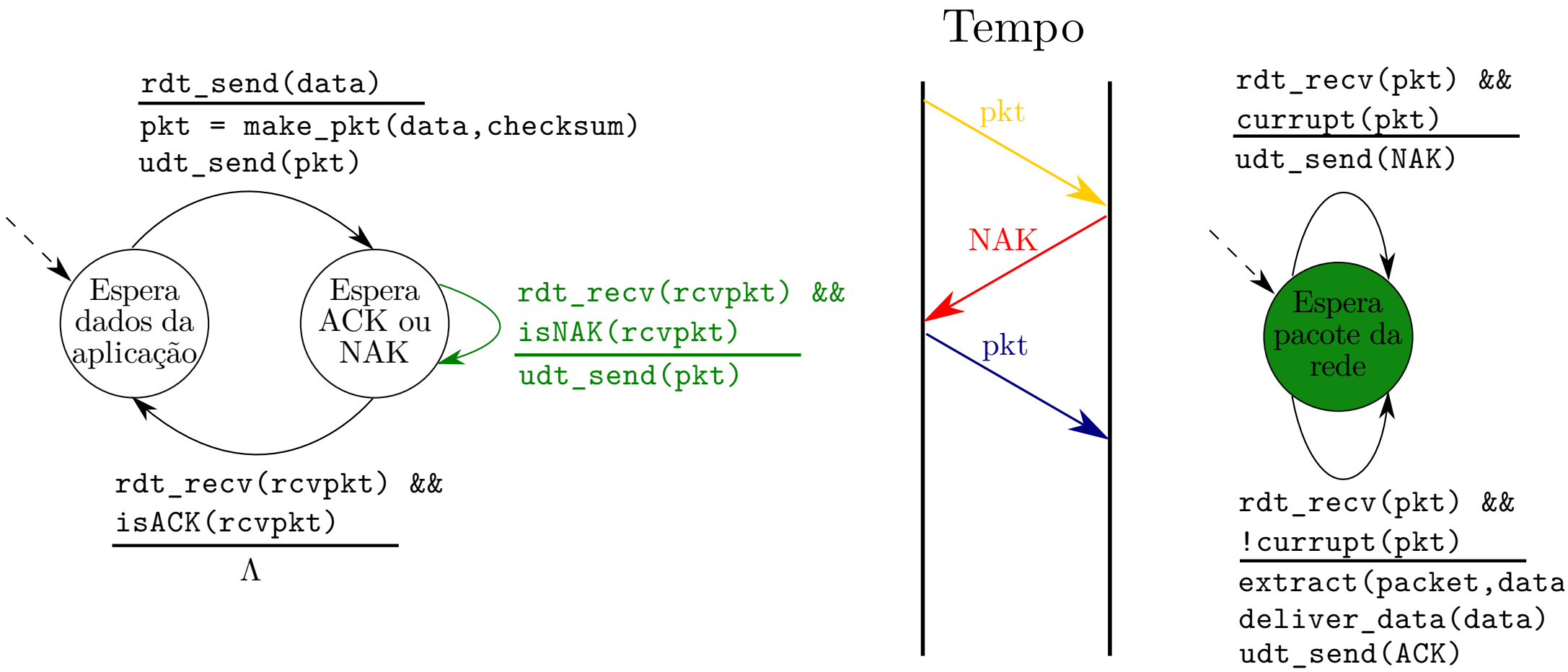




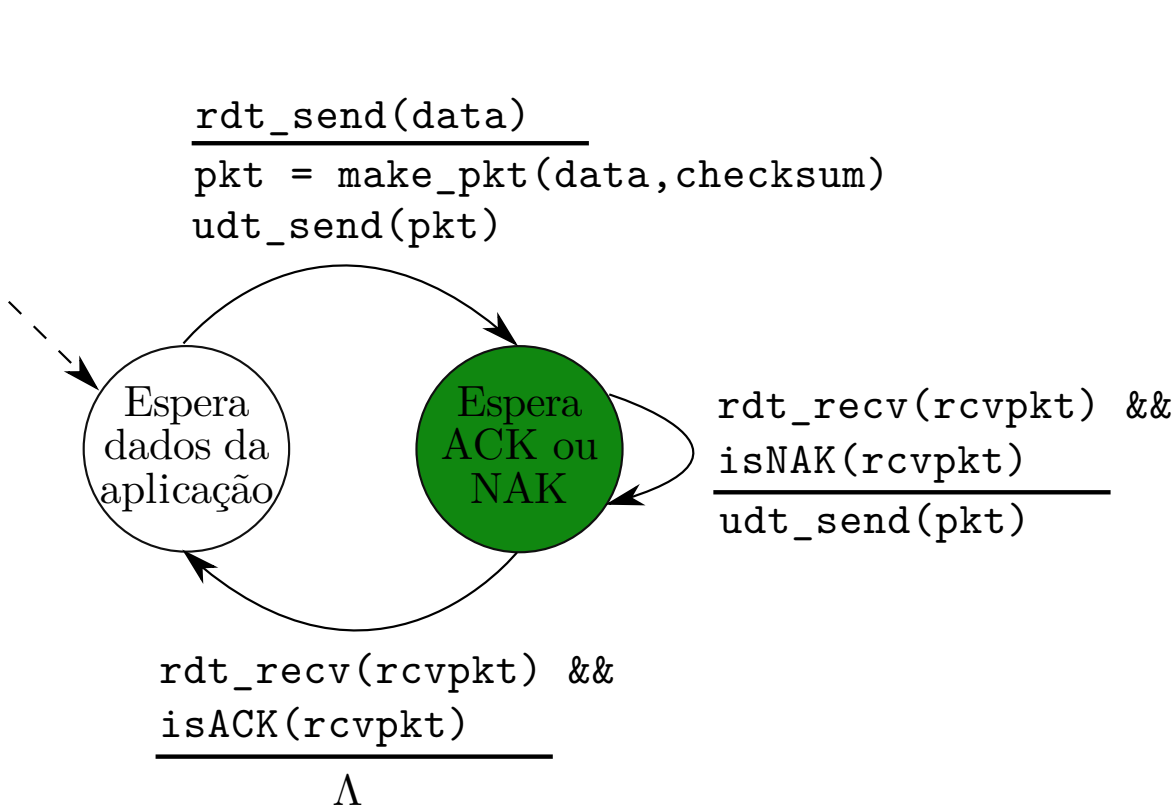
rdt2.0: Operação Com Erros (III)

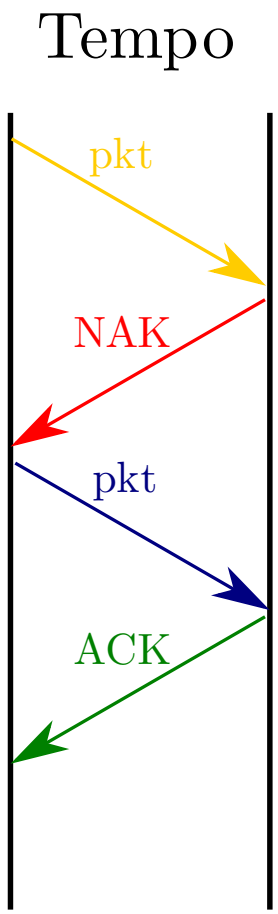


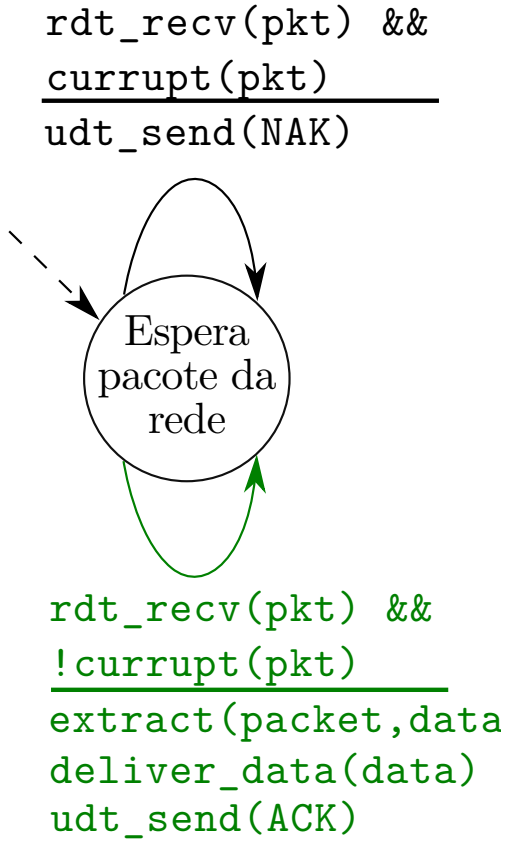
rdt2.0: Operação Com Erros (IV)



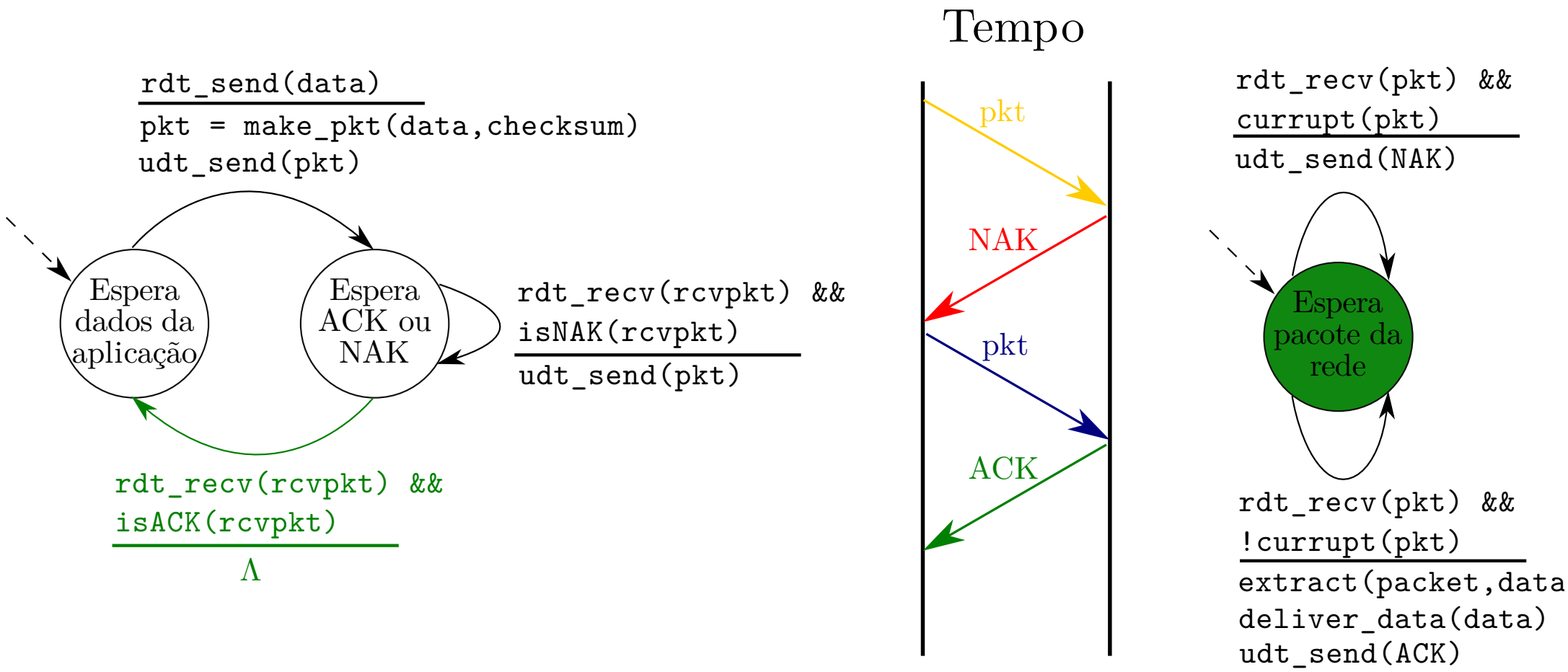
rdt2.0: Operação Com Erros (V)



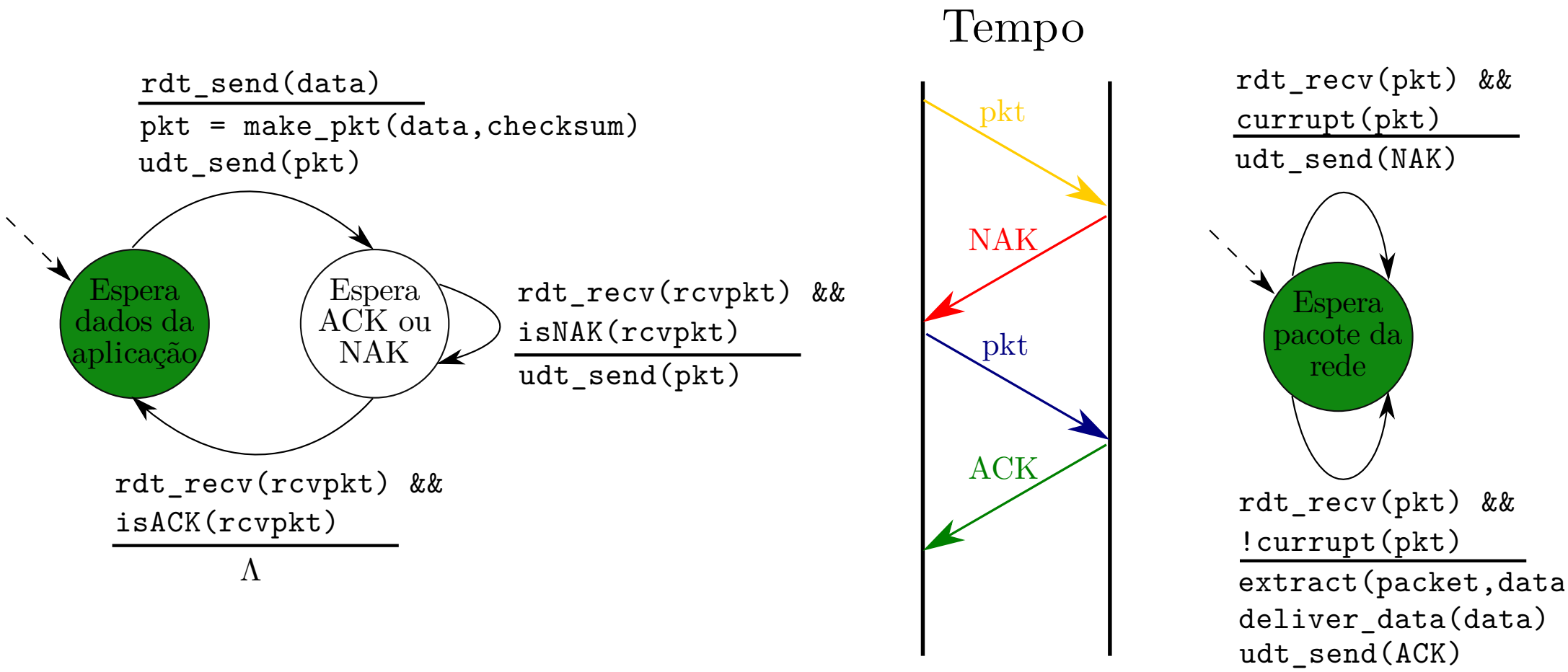




rdt2.0: Operação Com Erros (VI)



rdt2.0: Operação Com Erros (VII)



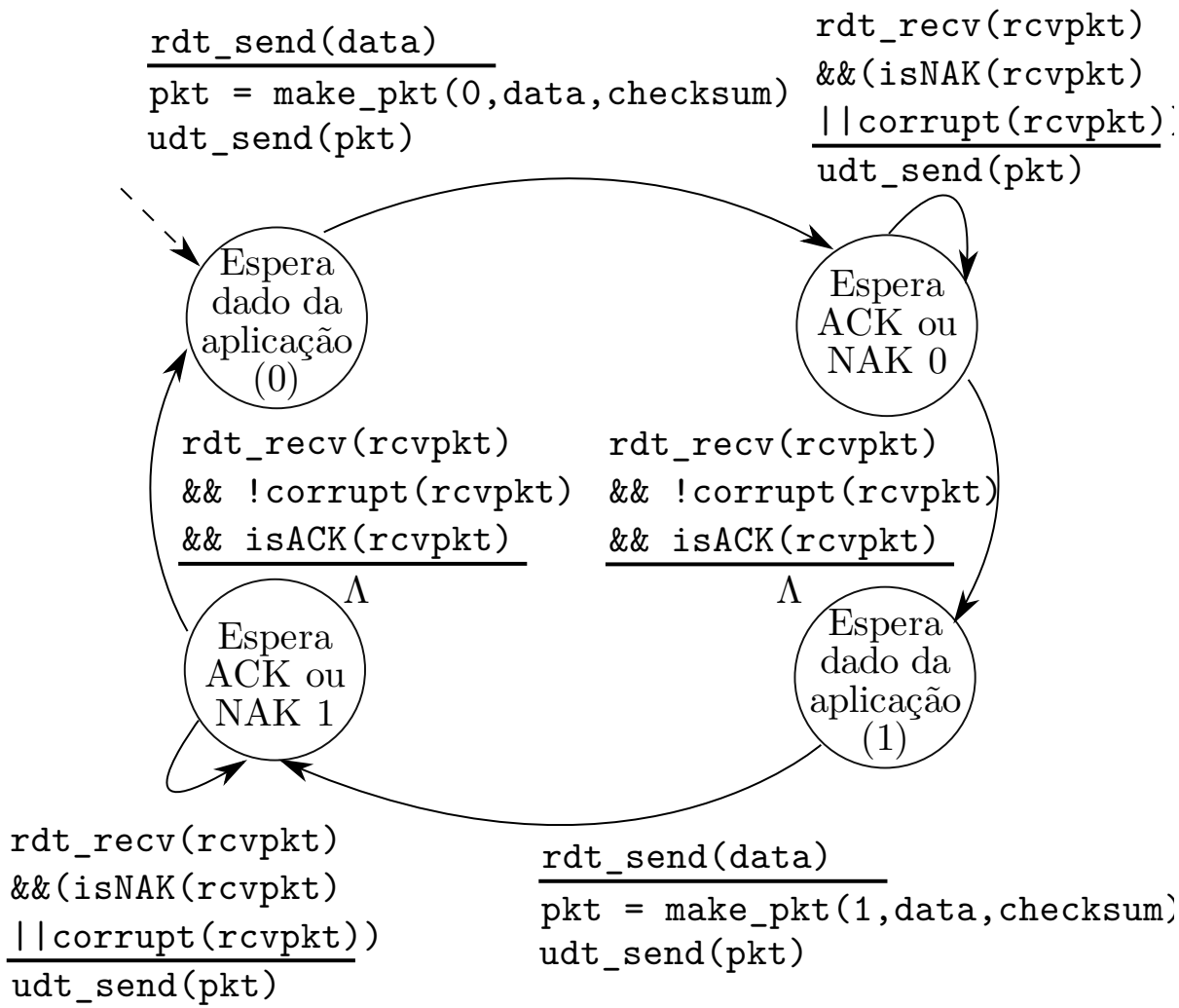
rdt2.0: Uma Falha Fatal!

- **O que acontece se ACK/NAK são corrompidos?**
  - Transmissor não sabe o que ocorreu no receptor!
  - Não pode simplesmente retransmitir: pode gerar duplicatas.
- **Lidando com duplicatas:**
  - Transmissor retransmite pacote atual se ACK/NAK é corrompido.
  - Transmissor adiciona um **número de sequência** a cada pacote.
  - Receptor descarta (não entrega à aplicação) pacotes duplicados.

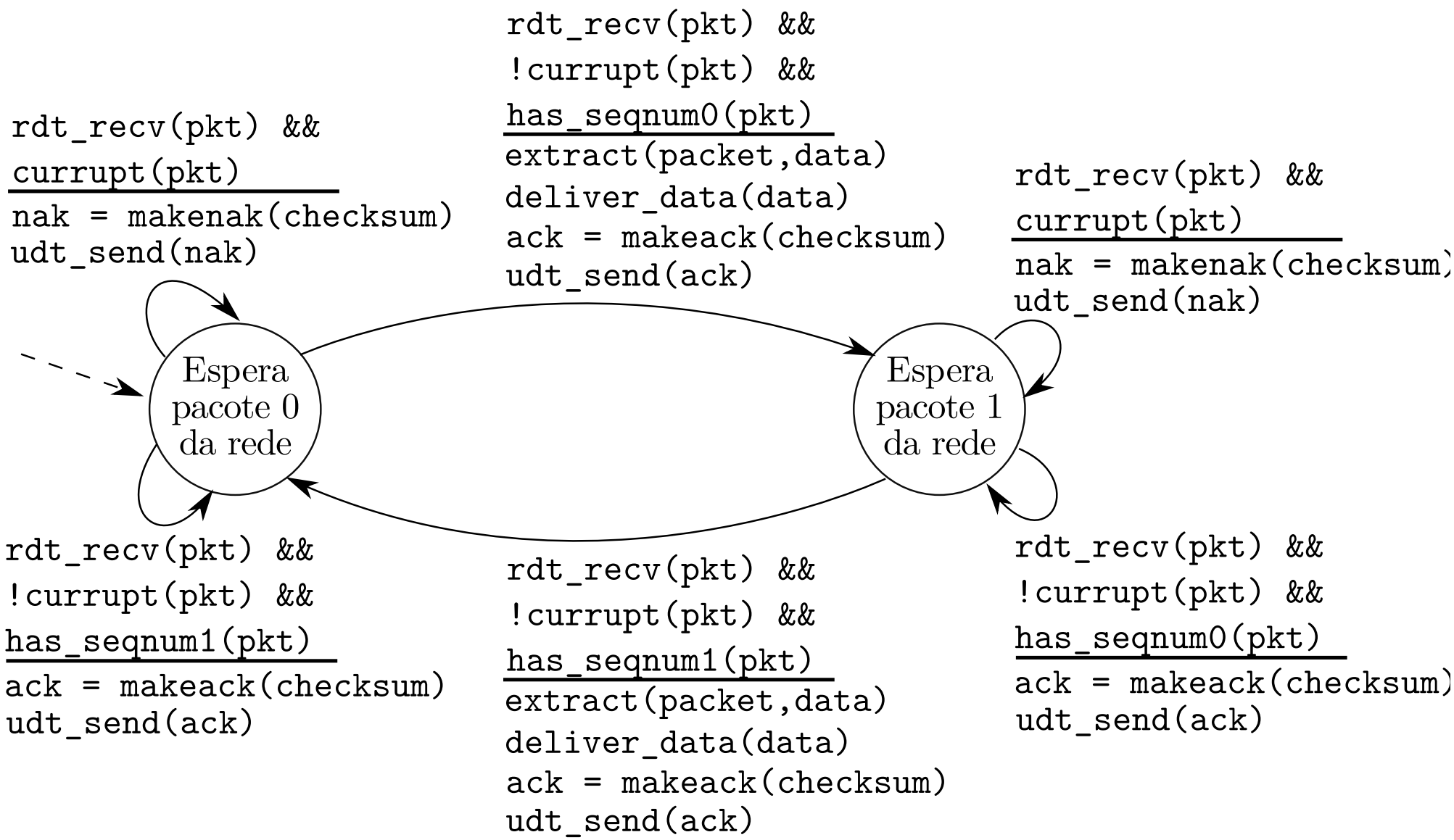
Stop and wait

Transmissor envia um pacote, espera pela resposta antes da próxima transmissão

rdt2.1: Lida com ACK/NAK Corrompido (Transmissor)



rdt2.1: Lida com ACK/NAK Corrompido (Receptor)



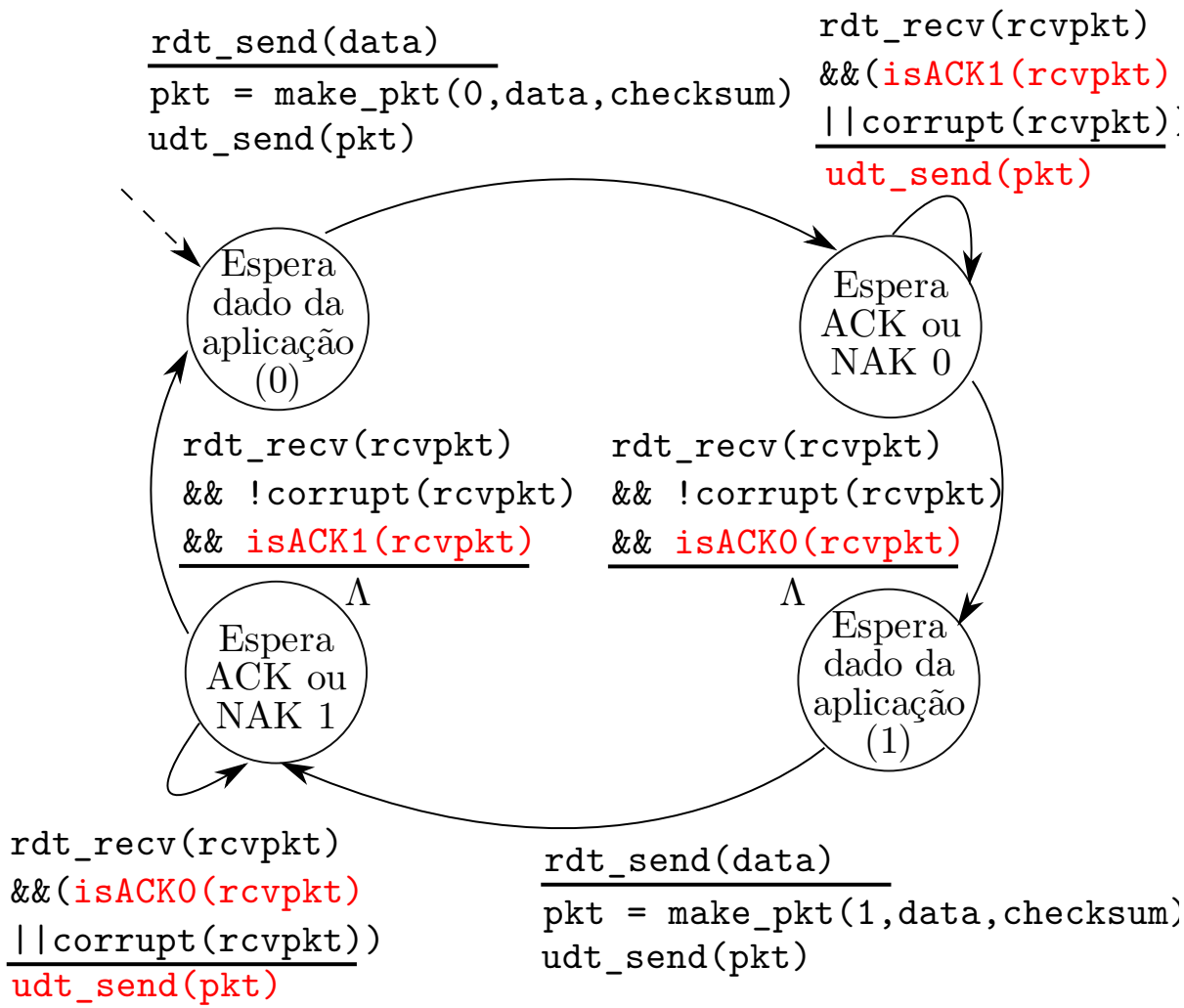
rdt2.1: Discussão

- **Transmissor:**
  - # de sequência adicionado a pacotes.
  - Dois valores (0 e 1) bastam. Por quê?
  - Precisa verificar se ACK/NAK recebidos estão corrompidos.
  - Duas vezes mais estados.
    - Estado "lembra" se # de sequência esperado é 0 ou 1.
- **Receptor:**
  - Deve verificar se pacote recebido é duplicado.
    - Estados indicam se pacote esperado é o 0 ou o 1.
  - Note: receptor não tem como saber se último ACK/NAK enviado chegou corretamente no transmissor.

rdt2.2: Um Protocolo Sem NAK

- Mesma funcionalidade do rdt2.1 usando apenas ACKs.
- Ao invés de um NAK, receptor envia **ACK para o último pacote recebido corretamente**.
  - Receptor precisará incluir no ACK **explicitamente** o # de sequência do pacote reconhecido.
- ACK duplicado no receptor resulta nas mesmas ações que um NAK: retransmitir pacote corrente.

rdt2.2: Transmissor



rdt2.2: Receptor