Network Congestion Control at the Application Layer

Paul Adamczyk, Federico Balaguer, Munawar Hafiz, and Craig L. Robinson

August 12, 2007

Abstract

Application-layer protocols play a special role in network programming. Typical programmers are more familiar with them and more likely to implement them. Well-designed application-layer protocols follow many patterns that improve the performance of applications using these protocols. We present a subset of these patterns that focuses on the congestion control at the application layer.

Introduction

Congestion is one of the main problems of networks. Congestion can lead to bottlenecks, which result in packet drops. Under the current paradigm of reliable network communication, dropped packets force applications to retransmit messages. Typically, congestion is battled at the network layer by adding more hardware or implementing better algorithms for handling individual packets.

However, low-level communication protocols have limited knowledge of the applications they serve. They can only be optimized for all types of traffic and cannot take advantage of characteristics specific to certain types of applications. Some tasks can be accomplished only at the application layer. The end-to-end argument [22] states that only the end-points of a communication can perform all functions related to the communication. This argument applies to network congestion as well. While much of the improvement comes from low-level infrastructure, only the applications can determine when to make some high-level adjustments to the communication protocol they are using. Because they understand the details of the application, application developers can design messages that use fewer resources thus limiting network congestion.

This paper presents a collection of patterns for reducing network congestion of point-to-point messages at the application layer. All the patterns share the same problem:

Network bandwidth is a scarce resource. It needs to be preserved also by application developers. What can be done at the application layer to limit network congestion?

Recently, there has been a revived interest in designing application-level protocols, e.g. for Web services. Unfortunately, many of the new protocol designers are unaware of existing solutions and tend to rediscover their inferior substitutes. This paper presents the best practices distilled from existing application-layer protocols and other systems for the benefit of the designers and implementers of new application-level protocols.

Network congestion affects client-server and peer-to-peer systems alike. We use the terms *sender* and *receiver* to describe the two parties involved in a message exchange, because they are more general. Moreover, since every message requires a confirmation/response, both the client and the server act as a sender and a receiver at some point, so thinking about them in terms of senders and receivers is simpler.

Sending messages over the network involves both *channel coding* and *source coding*. Since this paper is concerned with application-level protocols, only source coding solutions are relevant.

Network programming requires effective use of the underlying infrastructure, including other protocols used by the application-level protocols. Network communication results in overhead, both in data (additional message headers) and in messages (connection setup and teardown). Some data and messages are considered overhead by one protocol, but not by the underlying protocols. Limiting network congestion requires understanding how the underlying protocols work. For example, sending a message over TCP (without timestamp) over IPv4 and Ethernet without 802.1q produces 78 bytes of overhead per packet [8] as shown in Table 1.

Protocol	**Header Size** (in bytes)	**Max. Payload** (in bytes)
Ethernet	38	1500
IPv4	20	1480
TCP	20	1460

Table 1: Network Overhead Example

Selecting the most appropriate solution requires taking into account many conflicting forces and finding a balance between them. The key forces to consider while selecting the most efficient manner of congestion control are the run-time changes (understanding how the protocol changes, changes in message sequencing, timing, latency, and throughput as well as the resulting change in the system's performance), and design/re-implementation effort required to introduce the pattern.

Protocol Efficiency The solution cannot needlessly complicate the communication protocol. Replacing the existing protocol with too few large messages or too many small messages is likely to make the new protocol less efficient and less reusable.

Sequencing Any optimization must guarantee that the data is *processed* in the same order as before. This must hold true regardless of whether the data is sent or received in the same order as before–the sender and the receiver must cooperate to ensure the proper processing order.

Timing/Latency The optimizations cannot affect the timing of message exchanges. For example, it is not acceptable to delay sending a message until there is enough data to fill up the message payload to its limit. As multiple message exchanges are collapsed into fewer or one, the latency of a single exchange may increase.

Performance Congestion control is a valid concern to the application only if it improves the application's performance. An altruistic application would need to cease sending any messages, because this would result in lowest congestion, but this is not reasonable. The primary goal of every application is to perform its tasks as best as it can. A congestion-battling mechanism that diminished the application's overall performance is not acceptable.

Code Simplicity Optimizations typically produce more complex solutions than the simplest possible implementation. Any solution must consider the complexity of the code required to implement

it. If the code for producing and consuming messages is overly complex, which results in slow or erroneous execution, the network elements, rather than the network, will become the bottleneck.

The solution is to simply **send less stuff**–fewer messages and less data. Sending fewer and/or shorter messages that accomplish the same tasks is likely to decrease network congestion. To send fewer messages, it is necessary to define less verbose communication protocols that cut down the number of overhead messages. To send less data, the duplicate and unnecessary data needs to be eliminated.

Table 2 lists some of the possible techniques. The remainder of this paper presents patterns that explain these techniques in more details.

To mitigate network congestion:	**How? (pattern number)**
Send fewer messages	combine multiple messages (1)(2) short-circuit protocols (1)(3) piggybacking (5) use message throttling (6)
Send fewer overhead messages	long-lived sessions (4)
Send less data	eliminate duplicate data (2) compress the data (7) (8) send only changes from the previous value of the data (8) send data only if needed (3)
Send less overhead per messages	combine message payloads (1) send only changes from previous headers (8)

Table 2: Summary of network congestion solutions. Patterns describing them are listed in parentheses.

The patterns discussed in this paper include:

- Command Bundle
- Message Dispatcher
- Conditional Message
- Persistent Connection
- Piggybacking
- Self Throttling
- Data Compression
- Delta Encoding

A note on the synthesis of form:
The patterns described in this paper share many elements. To avoid repetition, the context, the problem, and the forces are described once. The description of each pattern begins with the solution, followed by the resulting context, known uses, and related patterns.

1 Command Bundle

When many small messages are exchanged, **combine a sequence of messages to the same recipient into a single message**. This increases the payload-overhead ratio, thus decreasing the use of network bandwidth.

Make each client encode multiple commands into the same network package. Figure 1 shows the Command Bundle architecture. The sender sends a collection of commands to the receiver using one network packet. Commands inside a network packet are separated by a terminating character. The receiver processes each command individually and responds accordingly.

Implementing the Command Bundle requires modifying the receiver, the sender, and the message structure.

- **Sender**. The sender packs multiple commands together into a single packet. It must ensure that they are placed in order and fit within the allowable packet size. Tight packaging of commands poses a problem when command producers stop generating new ones. If commands are not promptly available, less-than-full package should be sent. A "send it now" command should be available to command producers so that they can indicate that no more commands will be generated until the last ones are sent.
- **Receiver**. The receiver implementation has to consider that one network packet can hold multiple application messages and that some messages may span multiple packets. This requires implementing a command parser which has the capacity to split and join commands as appropriate. The parser must ensure that commands are delivered one at a time and in the order intended by the sender. One way to view this new design is to decouple the processing of commands from the mechanism that consumes messages.
- **Message Structure**. Multiple commands are combined into one larger packet and separated by a delimiter token.

This solution is applicable when the following conditions apply:

- You have the authority to modify the protocol (i.e. ordering and structure of exchanged messages).
- The time restrictions for delivering and processing messages are not strict.
- The size of a network packet is at least twice the expected size of a command.

* * *

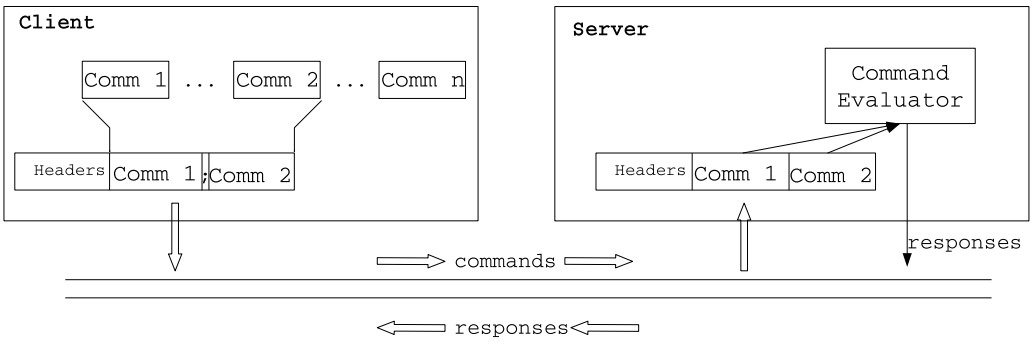


Figure 1: Architecture Supporting Command Bundle

Resulting Context

Protocol Efficiency Payload-overhead ratio of network packets is reduced, and fewer packets are sent.

Sequencing Commands are executed by the receiver in the same order they were encoded by the sender.

Timing/Latency Delays are incurred while composing and parsing a packet with commands. Message transmission latency is reduced due to channel loading. Once the packet has arrived, processing of messages contained in a packet is faster than if they were received one command per packet.

Performance The performance will improve if the overhead time (to construct packets of messages and produce results) is smaller than the time to build and reconstruct the same messages individually. If the network is not the bottleneck, this solution is likely to increase the time to execute the commands.

Code Simplicity Packaging ability must be added to the sender and parsing ability to the receiver. Additional buffer space may be required to store commands waiting to be processed.

* * *

Known Uses

Extended SMTP

One of the extensions to the Simple Mail Transport Protocol [9] is "command pipelining" defined by RFC 2920 [12]. Some SMTP commands such as: `RSET`, `MAIL FROM`, `SEND FROM`, `SOML FROM`, `SAML FROM`, and `RCPT TO` can appear anywhere in a pipelined command group. In this manner, multiple commands can be contained in a single packet.

Other commands: `EHLO`, `DATA`, `VRFY`, `EXPN`, `TURN`, `QUIT`, and `NOOP` can only appear as the last command in a group since their success or failure produces a change of state which the client SMTP must accommodate. These commands represent the "send it now" functionality of the sender.

Relational Databases

Two database managers provide solutions based on the Command Bundle: Informix [2] and Sybase.

The documentation usually refers to this solution as "Multiple Statements" or "Statement Batches". One of the problems found in the area of databases is that not all vendors support this feature and not all drivers support the handling of multiple SQL statements in one string.

TCP connection termination [17]

The connection termination phase uses a four-way handshake, with each side of the connection terminating independently. When an endpoint wishes to stop its half of the connection, it transmits a FIN packet, which the other end acknowledges with an ACK. Therefore, a typical teardown requires a pair of FIN and ACK segments from each TCP endpoint.

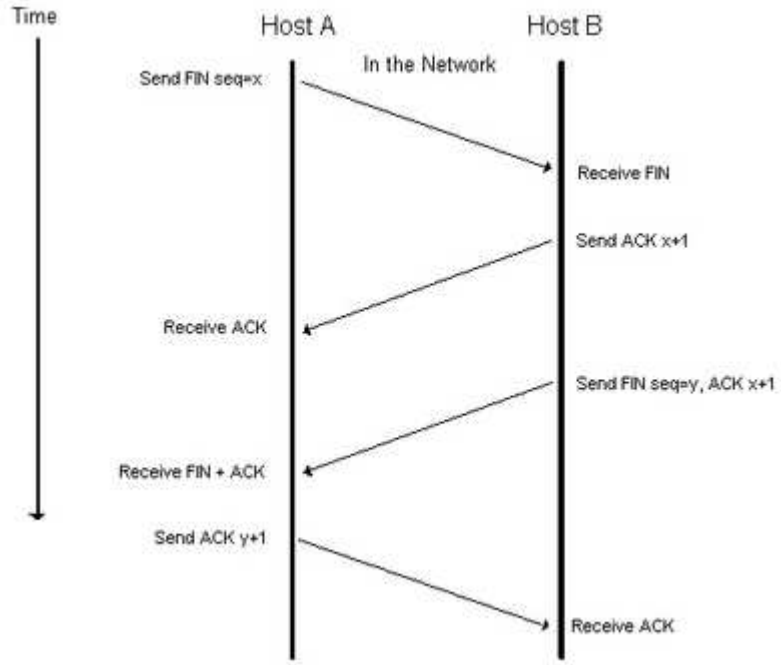


Figure 2: TCP connection termination

It is possible to terminate the connection by a three-way handshake, when host A sends a FIN and host B replies with a FIN & ACK (combines two steps into one message) and host A replies with an ACK.

Related patterns

Message Dispatcher (2) also combines multiple messages into one. Moreover, it eliminates duplicate data within the combined message.

2 Message Dispatcher

If multiple communicating systems transmit the same/similar data and domain-specific features allow for efficiencies to be achieved in a transmitted message, **define a separate entity that sends/receives messages on behalf of these systems, the message dispatcher**.

Provide a single interface which handles message delivery and sending for systems. Have systems subscribe to the Message Dispatcher and specify their communication requirements. Dynamically compose messages with a minimal information set so as to meet the requirements of all registered systems. Minimize the communication overhead by sending only one instance of the same data and packaging data more efficiently. At the receiving Message Dispatcher, interpret and demultiplex messages for distribution to subscribed systems.

The design is naturally decomposed into an *ontology* of data fields that are to be transmitted, and stipulating how the fields are to be composed into a message. Undefined data fields can be incorporated into the message by using a XML type scheme. Information can also be requested by one Message Dispatcher from another so as to meet the requirements of its subscribed systems.

Consider for example two pieces of data that are often sent at the same time. Rather than defining each one of them as a separate data field, define a single combined complex field, thus eliminating the overhead of multiple data headers. The dispatcher may also modify the data it sends. Rather than send all the data, send only small updates regarding the amount of change in the data from the previous message. This is more compact, thus reducing congestion.

Figure 3: Like in Noah's ark, Message Dispatcher has only **one** instance of each data type.

Implementing the Message Dispatcher requires adding the receiver and sender Message Dispatchers, and defining new message structure.

- **Sender**. Systems register with the Message Dispatcher and submit their data requirements. The requirements may include latency and frequency of transmission. The sender Message Dispatcher may obtain data in a variety of ways such as polling applications, maintaining a cache or generating the data by itself. The sender Message Dispatcher composes a message that meets all the requirements of the subscribed systems. Efficiencies such as removing duplicate data fields and combining related data into a single data field are performed. The message is dispatched.
- **Receiver**. Systems expecting to receive data register with the receiver Message Dispatcher. On reception of message, the receiver Message Dispatcher creates multiple messages, one for each subscribed system. The messages are then delivered to each system.
- **Message Structure**. Messages in transmit contain a union of the multiple sender system requirements. Consequently, the receiver does not know which application on the sender side generated

the data.

This solution is applicable when the following conditions apply:

- Multiple systems send and receive messages.
- Distributed coordination between multiple systems on the sender and receiver is prohibitively complex.
- Data is not unique to a particular system, but rather to the collection of systems on the sender or receiver.
- There is duplication of transmission data requirements between systems.

The Message Dispatcher is best applicable when messages are exchanged frequently in a broadcast fashion and when it is necessary to support adding new types of data and new systems. Moreover, it is best when the Message Dispatcher obtains the data independently of the subscribed systems.

* * *

Resulting Context

Protocol Efficiency Communication overhead is reduced, because information can be encoded in a single message in which duplicate fields are sent only once. Consequently, fewer messages are exchanged making the protocol simpler.

Sequencing The order of messages exchanged by Message Dispatchers does not change. But each Message Dispatcher can create local messages for its subscribers in arbitrary order thus making sequencing non-deterministic from the perspective of any single subscribed system.

Timing/Latency The overall time for a complete exchange can fluctuate, depending on the size of exchanged messages. The latency of each message increases, because each message needs to pass through *two* Message Dispatchers.

Performance If the time to (de)multiplex messages is low, the performance increases significantly, because less data needs to be transferred. The throughput of each message increases, because all the duplicate data is eliminated from the transmission. The performance gain increases as more communicating systems with overlapping data requirements register with the Message Dispatcher.

Code Simplicity The code required to implement the Message Dispatcher is not complex and amounts to finding a minimal set of information to be composed into a message. Construction and interpretation of messages according to the Message Composition rules requires simple passing of the message. The Message Dispatcher sits above the communication stack and thus the channel is managed by lower levels. Hence, implementing the Message Dispatcher only affects the systems which are registered with it. The potential complexity is to modify existing systems so as to register and stipulate information requirements with the Message Dispatcher.

An additional result:

Extensibility Disparate systems on a different peers can communicate through a standard communication interface. In this way the Message Dispatcher acts as Facade [14] for multiple communicating systems. This enables new systems to be developed without dependence on existing systems and without requiring additional messages be created. In other words, the mechanics of the applications are separated from the communication and information sharing concerns.

* * *

Known Uses

Collaborative Inter-vehicle Wireless Safety Applications [20]

In this example, vehicles communicate with each other using the wireless channel and share information on road, vehicle and driver conditions. Several applications have been developed including emergency brake warnings, traffic light violation warnings or detecting collisions. Although the applications are different, many of the data fields required are common, e.g. vehicle position and speed.

With the potential for many vehicles to be transmitting simultaneously, and the difficulty associated with coordinating transmissions), reducing channel load so as to reduce interference is important.

The Message Dispatcher has been deployed on several test vehicles at the Toyota Technical Center in Ann Arbor, Mi. The concept has also been adopted in the Society of Automotive Engineers (SAE) standard for inter-vehicle wireless communication. It has been found to be particularly useful in adapting to the changing specifications and requirements of the deployed applications by essentially decoupling the mechanics of the communication policies.

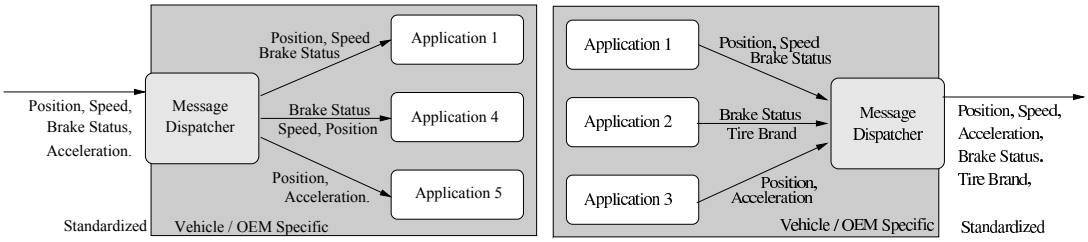


Figure 4: The Message Dispatcher assimilates data requirements from all the on-board applications and compiles a single message using a dictionary of defined data elements and standardized message construction guidelines. A receiving Message Dispatcher is responsible for separating and disseminating data elements from the received message to all on-board applications as well as managing data requirements for surrounding vehicles.

Facebook

This social networking website keeps subscribers informed of their associates' activities. When two colleagues perform a similar action, (e.g. both join a group or become friends with someone) a single notification is provided to their associates. For example, instead of two separate notifications–"Craig wrote a patterns paper" and then "Paul wrote a patterns paper"–a single notification "Paul and Craig wrote a patterns paper" would be shown to all subscribed parties.

Related patterns

Command Bundle (1) combines multiple messages, but it does not eliminate duplicate data from multiple messages.

Message Dispatcher described in the Enterprise Integration Patterns book [16] differs from this pattern. It provides only message dispatching based on the recipient. It does not consider the contents of the message and cannot recreate multiple messages for different recipients from a single message.

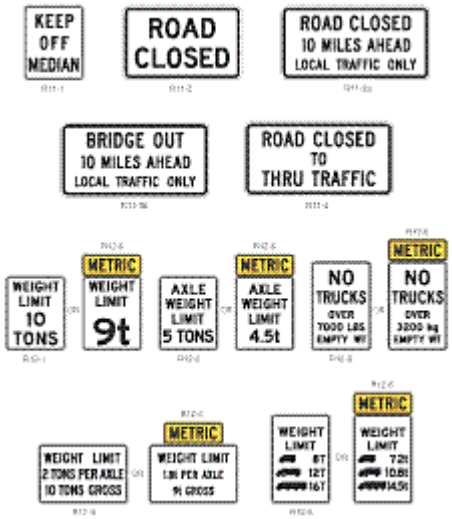
Publisher-Subscriber [6] is an alternative way of disseminating changing information for the benefit of a large number of recipients. The Publisher sends out the information that changed and the Subscribers-recipients are notified of that fact. The Message Dispatcher plays the role of the Publisher when it is sending out a new message to other Message Dispatchers. It also plays the role of the Subscriber when it receives a message, updates its own state, and passes the newly acquired information to the local communicating systems that are registered with it.

The main benefit of Message Dispatcher over Publisher-Subscriber is that it minimizes the number (and size) of messages sent between communicating systems.

3 Conditional Message

When many messages are exchanged to negotiate the optimal type of the response, **provide enough data/context in the request so that the recipient can immediately determine what data to send back**.

To ensure that both the sender and the receiver agree on a specific course of action (e.g. selecting a particular response from several alternatives), they often need to exchange multiple messages. Sometimes, for example when setting up a connection (i.e. handshaking), such long exchanges are necessary. But once a connection is established, message exchanges that follow should be simplified whenever possible, because they are costly in time and processing. By providing enough relevant information (typically metadata) in the initial request, the sender may be able to reduce the number of exchanges. Similarly, the receiver can reduce the number of exchanges by guessing the expected result based on its knowledge of the sender.



Implementing the Conditional Message requires modifying the receiver, the sender, and the message structure.

- **Sender**. The sender constructs a message and considers any potential information expected in a subsequent response from the receiver. This information is included in the request, provided the additional data overhead is not prohibitively large. This, a conditional message is generate based on the receiver's expected response and a message construction policy of the sender (e.g. maximum likelihood).
- **Receiver**. Considers contents of *entire* received message as well as its own capabilities before constructing a response and dispatching it to the sender.
- **Message Structure**. The message contains a set of conditional statements (or equivalently an ordered list of preferences) related to the receiver's expected responses.

This solution is applicable when the following conditions apply:

- The next step in the protocol is determined based on the state of the sender and receiver.
- The replaced message exchange is *not* used for establishing the communication between the sender and receiver, i.e. there is already an active connection between them.

* * *

Resulting Context

Protocol Efficiency The number of message exchanges is reduced. The initial request message may be significantly larger (to account for all potential outcomes). The response may be as short as "no change."

Sequencing By reducing the length of the negotiation sequence, the complexity due to sequencing is reduced.

Timing/Latency The time of the complete exchange decreases, because fewer messages are exchanged. However the latency of an individual response increases. This is because the conditional message is longer, requiring greater generation and processing time. Alternatively, the recipient may need to perform extra processing to determine how to respond.

Performance If there are many conditions to consider in a sequence, the performance may be greatly enhanced, because this solution decreases the communication time, the processing time, and the size of exchanged data. As a result, the message throughput may decrease.

Code Simplicity The resulting code is more complex at possibly both sender and receiver side, because they must consider all combinations of possible outcomes.

* * *

Known Uses

HTTP's Conditional GET [11]

HTTP supports three ways to check if a representation of a resource stored by a client is still up to date. The client could send a HEAD to request the metadata of the resource from the server. If the server returns newer metadata than what the client has, the client sends a GET to get the new contents. This means that two messages are exchanged. Alternatively, the client could request data by sending a GET request. The server would respond with the current resource representation, which may be the same as the data already held by the client. A better solution is to send a conditional GET message–the same as regular GET, but with a conditional header (e.g. If-Modified-Since, which contains the timestamp of the client's current version of the resource). If the server has a newer version, it sends the data; otherwise, it responds with the status code "304 Not Modified." Only one message exchange is needed.

HTTP Content Negotiation [11]

Content negotiation enables clients and servers to determine the optimal format of a resource (e.g. GIF, JPG or PNG for a picture). The server stores the resource internally in a specific format. Each client (called *agent* in HTTP specifications) preferrs certain formats. HTTP defines two types of content negotiation–server-driven and agent-driven.

The server-driven negotiation means that the algorithm for selecting the best representation of a resource is located on the server. The client specifies the preferred format of a resource in HTTP headers (e.g. Accept, Accept-Language, Accept-Encoding) of the original request. The server decides what is the optimal format by taking into account the preferences of the client. Only one message exchange is needed.

The agent-driven negotiation gives the agent more control over the representation. First the client submits a request for a resource (with or without a list of preferences). The server responds with the status code "300 Multiple Choices" that includes a list of available representations. The client selects, either automatically or with user intervention, the most appropriate format and requests it again from the server. This approach results in sending the best possible match to the client, but it requires two message exchanges.

VNC

VNC (Virtual Network Computing) protocols support many formats of data transmitted between the client and the server. For example, in the RFB (remote framebuffer)[19] protocol, the data passed between the client and the server represents pixels on the client's screen. RFB supports many formats, so each client-server pair can negotiate their own preferred format. Rather than suggest one format per message, the client includes the list of all encodings it supports, in the order of preference, in a `SetEncodings` message. The server can use any of the requested formats, but it may also ignore client's preferences and select a *raw* format of the response, which is the default that all RFB clients must support.

Related patterns

Command Bundle (1) combines a sequence of messages into one, but it cannot collapse messages into fewer commands. The receiver applies the commands sequentially. In the Conditional Message, the next message in the protocol depends on the context (e.g. the results of the previous messages).

4 Persistent Connection

When the connection setup (and/or teardown) requires multiple message exchanges, **establish the connection only once and do not tear down the message channel[16]. Keep an open connection and reuse it for sending subsequent messages**. This produces a dedicated channel between the sender and the receiver that is used for multiple message exchanges.

Persistent connections can also support *pipelining*, i.e. the ability to send multiple requests without waiting for a response to the previous requests.

Implementing the Persistent Connection requires modifying the receiver, the sender, and (in some cases) the message structure.

- **Sender**. The sender initiates connection setup. Once the connection is established, the sender monitors its state to determine if the connection is still open when a new request is to be sent. The sender can monitor the status of the connection and close it if it's no longer required.
- **Receiver**. The receiver acknowledges when the persistent connection is established. If it determines that no more communication will occur on this connection, the receiver may optionally include an indication that the connection is closing in the response, and then close the connection once the response is sent.
- **Message Structure**. In many cases, the message structure is exactly the same as if it were a standalone message with its own connection. However, when the sender *or* the receiver decides to close the persistent connection, the message will include an indicator of the connection state *after* this message is consumed. A new message type may be needed for maintaining the connection status.

This solution is applicable when the following conditions apply:

- Establishing a communication channel between the sender and the receiver requires some negotiation or handshaking
- The communication between the sender and the receiver includes more than one message exchange
- Connection maintenance requires less overhead than establishing the connection every time

* * *

Resulting Context

Protocol Efficiency After the connection is established, subsequent exchanges do not require overhead messages, hence fewer overhead messages need to be sent. However status update messages do

require some overhead, but may be mitigated using piggybacking. The application protocol does not change.

Sequencing Once the connection is established and pipelining is used, multiple requests will be sent without waiting for the response.

Timing/Latency Once the connection is established, subsequent responses are received faster than before, since there is no individual connection setup overhead. The latency of a single message is unchanged.

Performance The benefits of persistent connection are proportional to the number of exchanged messages. If a typical message exchange consists of only one request-response pair, persistent connections unnecessarily waste resources.

Code Simplicity Code changes can be localized both on the sender and the receiver, potentially to a single conditional check for whether an active connection for the sender-receiver pair exists.

* * *

Known Uses

HTTP/1.1 [11]

HTTP messages are transmitted over TCP. To send an HTTP message, it is necessary to set up a TCP connection (which requires 4 messages). In HTTP/1.0 [5] the connection is closed after each message exchange (closing the connection requires sending 3 or 4 more TCP messages). This is very inefficient. Many performance studies (e.g. [18]) have shown this. Downloading a Web page consisting of 10 resources requires 90 TCP messages (setup: 4, payload: 2, teardown: 3, repeated 10 times). By keeping the TCP connection open, downloading 10 resources requires 27 TCP messages (setup: 4, payload: 2 * 10, teardown: 3). In HTTP/1.1 all connections remain open unless explicitly closed. The HTTP systems indicate if the connection is closed by including the "Connection: close" header in the response.

HTTP/1.1 supports message pipelining–the client sends multiple requests without waiting for the response.

ODBC

Open Database Connectivity (ODBC) support persistent connections, with or without pipelining. For example, Microsoft SQL Server (starting with version 6.0 [3]) uses server-side cursors to support multiple outstanding requests on a single connection handle. Each cursor operation in the ODBC driver generates one individual cursor command which is sent to the SQL Server. When the resulting set for each cursor command is received by the client, the SQL Server accepts another command from another statement handle over that connection handle.

ATM

ATM networks use persistent connections. Resource reservation is done once and then no subsequent control messages are sent.

Related patterns

Message Channel [16] describes the details of implementing a dedicated channel between two entities. **Piggybacking** (5) can be used to include connection status data in existing messages.

5 Piggybacking

If two network elements are already engaged in message exchange, **pass new data, unrelated to the current exchange, in the current message exchange**. Include the new data as an addendum to the exchanged messages rather than sending a new message.

A long message exchange is often called a *conversation*. This solution suggests using payload capacity in existing conversations to facilitate opportunistic communication of unrelated data. This is especially effective if the exchanged messages have some spare space in the message payload where the new data can be added.

Implementing the Piggybacking requires modifying the receiver, the sender, and the message structure.

- **Sender**. When required to send a new piece of data to a receiver, the sender examines existing communication with that receiver. If there is one, the sender incorporates the data into the existing conversation.
- **Receiver**. The receiver monitors incoming messages to determine if new data has been appended to message associated with existing conversations.
- **Message Structure**. Messages intended for a single receiver may contain any number of unrelated data elements in addition to the ongoing conversation data.

This solution is applicable when the following conditions apply:

- Sender and receiver frequently conduct unrelated conversations
- The additional data to be sent is small in relation to the volume of the existing communication

* * *

Resulting Context

Protocol Efficiency The message recipient receives the data from multiple conversations in a single message, hence fewer messages are exchanged.

Sequencing The relative sequencing of messages between conversations is indeterminate. Typically this is not a serious concern, because piggybacking usually involves two unrelated conversations.

Timing/Latency Timing between receiving data which has been piggybacked is less predictable, since this data is opportunistically included in existing communications. Latency does not change for each individual message, unless there is a significant time delay due to (de)multiplexing data for each conversation.

Performance Starting a new message exchange does not require the connection setup time. As a result, the actual data is exchanged faster. Additional message overhead for the data that is piggybacked is also avoided. The throughput of the data-carrying messages may increase because of the piggybacked data or, if the additional data can be accommodated in the spare bits of the original communication, it remains the same.

Code Simplicity The improved performance comes at the cost of more complex code for both sender and receiver. The sender needs to account for many possible combinations of incorporating new data into existing messages. The receiver needs to demultiplex the data before passing the appropriate received data to different processing units.

* * *

Known Uses

SMS [13]

Cell phones use two dedicated channels for communication–a low-bandwidth control channel and a traffic channel for sending voice packets. Typically, simple commands (e.g. start ringing, user pressed the * button) are sent to/from the phone on the control channel, because it requires fewer resources. Text messages (SMS) are exchanged as new messages sent on the control channel. However, if the user is already in a phone call, there is a traffic channel for sending voice packets. These packets have enough spare bits to pass the extra data (the text message). By using the extra bits in the existing messages, no additional resources are used to send text messages.

Mobile phone voicemail notification [13]

In the ANSI-41 protocol for mobile phone communication, a notification message is sent from the system (specifically, HLR) to the phone when the subscriber receives a voicemail. But if the HLR and the phone are already exchanging another control message (e.g. updating phone's location, periodic authentication), the voicemail notification field is added to that control message. Thus, rather than sending another message, an existing message is used with few extra bytes appended.

TCP [17]

To ensure reliable service in TCP (Transmission Control Protocol), every request is acknowledged by the receiver with an acknowledgement number. In a communication between two peers, where both send requests and responses, the acknowledgement number of a prior message is included in the following request from the other peer.

Related patterns

Message Dispatcher (2) has more details on one way to implement demultiplexing of data in a message exchange.
Command Bundle (1) can be thought about as a form of piggybacking. If the first command in the bundle is the main conversation, all the subsequent commands are piggybacking on top if it.

6 Self Throttling

When sending messages is "expensive" for the sender, **adapt the frequency of message sends based on the significance of the transmitted message**. Enable the system as well as the users to set and modify guidelines as to when such messages must be sent out. The application chooses transmit times based on the existing conditions and imposed guidelines. The user should be able to override the application's decision.

The key to the solution is striking a balance between *expensive* and *interesting* messages. For example, if the sender does not receive a response in the expected time, resending the request becomes more "expensive," because it is more likely to time out again. Alternatively, if no new event occurs for a long period of time, the message containing the unchanged data is less "interesting" to the receiver, because it does not report any new information.

Implementing the Self Throttling requires only modifying the sender.

- **Sender**. The sender maintains some type of system state which it uses before sending a message, to determine how "expensive" the message is to send, and how "interesting" it is to the receiver. The sender decides to send the message, and then updates its history of adaptive predictions accordingly.
- **Receiver**. No change.
- **Message Structure**. No change.

This solution is applicable when the following conditions apply:

- Sending a message is "expensive."
- How "interesting" and "expensive" a message is varies.
- The application needs to adapt to changing network conditions.

* * *

Resulting Context

Protocol Efficiency Although there is no change in the protocol (the same messages are sent in the same order), the resulting message exchange is more efficient, because messages are sent only when required, based on the current system state. Hence, fewer messages are sent per unit of time.

Sequencing There is no change in the sequencing of the messages, only in their frequency.

Timing/Latency Messages are sent less often, so the relative time between messages increases. But the messages are still sent often enough when they become "interesting." Message latency remains unchanged.

Performance Since there are fewer messages, and the processing required to check whether to send a message is typically minimal, the overall performance of self-throttling systems increases. As less data is exchanged, the throughput typically decreases.

Code Simplicity Defining adaptive algorithms for adjusting the throttling parameters is often the most difficult part of implementing this solution. System state must also be maintained. Some effort is needed to add checks of the throttling parameters when a relevant event occurs. Some work is required to enable the user to change the throttling parameters.

* * *

Known Uses

Cellular telephony

Messages that are sent out periodically are considered "expensive" in cellular networks. They are sent only when an "interesting" event happens. For example, location updates are sent only if a cellular phone has moved from one location area (a group of cells) to another, or has not sent a location update nor made a call within a relatively long time (e.g. one hour). This helps to relieve congestion of the signaling channels in both North American and European cellular systems.

Ethernet

In Ethernet, only one node can be sending frames at a time. If a sending node senses that the communication channel is idle, it starts to transmit the frame. While transmitting, it monitors the presence of signal from other nodes. If the node transmits the entire frame without detecting signals from other nodes, the sending is complete. But if it detects signal from other nodes while transmitting, it stops transmitting the frame, and transmits a jam signal. After aborting, the node enters the *exponential backoff* phase–it waits to retransmit progressively longer as it encounters more collisions with other nodes.

Collaborative Inter-vehicle Wireless Safety Applications [20]

Recall that the basic functionality of the Collaborative Inter-vehicle Wireless Safety Applications was described in the Message Dispatcher pattern. Individual applications subscribed to the Message Dispatcher or the Message Dispatcher itself perform self-throttling by selectively excluding the data that is not sufficiently interesting. For example, an application monitoring a vehicle's location does not need to send updates to surrounding vehicles if the vehicle is moving in the same direction at the same speed. The location-monitoring applications of surrounding vehicles can calculate this information by themselves. If a vehicle's location changes in an unpredictable way (e.g. if it exits from the interstate), its location-monitoring application needs to send a message with the new location. The new location becomes "interesting" to the surrounding vehicles, because they cannot calculate it from their own data.

Content Distribution

In Content Distribution systems, not all mirrors get updated with the most recent version of the content simultaneously. Update algorithms vary, but the processing of the updates is throttled so that the mirror spends most of its processing handling requests rather than updating its contents.

7 Data Compression

Rather than passing data in its original format, **apply a compression algorithm to the message content before sending it.** Compression applies to the data as well as the message headers.

Popular data formats, such as XML and plain text can be compressed very effectively. For example, one military study [23] has found that XMill hybrid compression can reduce the size of large XML documents to 1% of the original size.

The *entropy* of a set of data measures the amount of randomness in the data [7]. For example, a bit sting of ones has an entropy of zero since there is no randomness - all the bits are 1. However, a string of random ones and zeros (e.g. outcomes of fair coin tosses) has entropy 1. In this way, entropy represents a bound on the potential effectiveness of data compression. For example, in order to uniquely identify 8 types of widgets, at least 3 bits are needed. More may be used, but despite one's best efforts to compress widget description data (e.g. running a zip application twice), the number of bits needed cannot be reduced below 3. Similarly, in the coin toss example, which has entropy 1, a single bit is required to describe each outcome of the coin toss. Thus, data compression is best suited for data with high entropy.

Implementing the Data Compression requires modifying the receiver, the sender, and the message structure.

- **Sender**. The sender compresses data before transmitting.
- **Receiver**. The receiver uncompresses the data upon receipt.
- **Message Structure**. The new message contains compressed data. Unless the compression algorithm is specified, the original data is not generally recoverable from this representation.

This solution is applicable when the following conditions apply:

- The data type has a suitable compression algorithm
- The time to compress and decompress the data is smaller than the time to transfer the data over the network in the original, non-compressed format

* * *

Resulting Context

Protocol Efficiency There is no change to the protocol.

Sequencing No change.

Timing/Latency Timing does not change either. If the compression shrinks the message to fit into fewer packets, then message latency decreases; otherwise it is unchanged.

Performance With compression, messages have smaller payloads. If the data spans multiple network packets, there are fewer packets to send per message. Hence, throughput is increased. Only if the (de)compression time is long, compared to the network speed does this solution degrade performance.

Code Simplicity Adding the code to implement each new data format is a one-time effort. Once implemented, it can be reused in other applications. Generic compression algorithms may also be used, e.g. Ziv-Lempel.

* * *

Known Uses

HTTP [11]

HTTP/1.1 supports end-to-end and hop-by-hop message compression. The end-to-end compression is specified in the Content-Encodings header. It describes what additional encoding of the message was performed by the sender (e.g. "Content-Encoding: gzip"). The hop-by-hop compression is specified in the Transfer-Encoding header (e.g. "Transfer-Encoding: chunked"). It is applied by the network elements transmitting the HTTP messages to speed up the transmission.

Data Aggregation in Sensor Networks [15]

In LEACH (Low-Energy Adaptive Clustering Hierarchy) wireless application-level protocol, nodes are organized in clusters. Only the cluster-heads communicate with the base station by sending and receiving messages. The data collected by individual nodes is passed to cluster-heads, which collects the data, analyzes it, and sends a single message to the base station. Rather than sending individual data points, the cluster-head sends a summary (e.g. a sufficient statistic, such an average value of all the nodes).

Web services: SOAP compression

Apache Axis/1.x supports compression of SOAP messages [4]. The metadata about the compression in use is sent from the client to the server in HTTP headers. The client compresses the SOAP request and indicates the compression algorithm in the "Content-Encoding" HTTP header. If the client is willing to accept a compressed response, it adds an "Accept-Encoding" header field indicating the acceptable encoding (typically gzip). If the response from the server includes the "Content-Encoding" header, the SOAP response is decompressed before being processed by the Web service.

Related patterns

Delta Encoding is a special case of Data Compression where only the information that changed from the previous message is included in the following message.
Message Dispatcher performs a simple type of data compression by eliminating duplicate instances of the same data field.

8 Delta Encoding

When the exchanged messages contain updates of data, **send the change from the previous value rather than the entire contents.** Delta encoding is applicable to the data as well as the message headers.

Implementing the Delta Encoding requires modifying the receiver, the sender, and the message structure.

- **Sender**. The sender calculates the changes of the data to be transmitted from the most recently sent update. It encodes the changes in the delta format and sends the message containing only the changes. Periodically, the sender sends the complete representation to enable the receiver to verify that its representation is still correct.
- **Receiver**. Upon receiving the delta message, the receiver applies the deltas to the data it contains. Upon receiving a full representation, the receiver checks it against its copy. If the representations are not identical, it replaces its current copy with the one received and (potentially) initiates some consistency checks to determine the cause of the discrepancy.
- **Message Structure**. Most messages contain only the changes from previous message. The original data is not recoverable from this representation; only the receiver that stores the previous values of the data can interpret the encoding.

This solution is applicable when the following conditions apply:

- The data transmitted in subsequent messages changes slowly
- The changes can be described in a succinct format
- The time to recover the original data is shorter than the time to transfer the data over the network in the original format

* * *

Resulting Context

Protocol Efficiency There is no change to the protocol.

Sequencing Sequencing does not change. It cannot change. Because messages contain only updates, processing them out of sequence may result in unpredictable behavior. Ensuring that messages are processed in the same order they were generated is critical.

Timing/Latency Timing does not change. Since the delta encoding shrinks the message to fit into fewer packets, message latency decreases.

Performance With delta encoding, messages have smaller payloads. If the data spans multiple network packets, there are fewer packets to send per message. Hence, throughput is increased. Because they only communicate with deltas, the sender and receiver need to maintain the actual state of all exchanged data and ensure that it is correct. If the storage requirements are high, the performance may decrease.

Code Simplicity As more advanced encoding algorithms are used, the efficiency of delta encoding increases, but the code complexity increases. Message data storage management may also become complex.

* * *

Known Uses

MPEG-2 [1]

MPEG is an encoding and compression system for digital multimedia content defined by the Motion Pictures Expert Group (MPEG). MPEG-2 video compression algorithm achieves very high rates of compression by exploiting temporal and spatial redundancy in video information. *Temporal redundancy* indicates that successive frames of video display the same scene. The content of the scene often remains fixed or to change only slightly between successive frames. *Spatial redundancy* occurs because parts of the picture are often replicated (with minor changes) within a single frame of video.

In addition to highly efficient compression, MPEG must enable random access to the video. To accomplish these two tasks efficiently, MPEG-2 supports three main picture types: I-Pictures, P-Pictures, and B-Pictures. Intra coded pictures (I-Pictures) are coded without reference to other pictures. They provide access points to the coded sequence where decoding can begin, but are coded with only moderate compression to take advantage of the spacial redundancy. Predictive coded pictures (P-Pictures) are coded more efficiently using motion compensated prediction from a past intra or predictive coded picture. They can be used as a reference for further prediction. Bidirectionally-predictive coded pictures (B-Pictures) provide the highest degree of compression, because their contents are based on differences from both past and future reference pictures. The organization of the three types in a sequence is left to the encoder and depends on the requirements of the application.

ROHC [10]

Robust Header Compression (ROHC) is a standard for compressing RTP/UDP/IP (Real-Time Transport Protocol, User Datagram Protocol, Internet Protocol), UDP/IP, and ESP/IP (Encapsulating Security Payload) headers. The ROHC algorithm is similar to video compression. The first packet sent is the base frame that contains complete headers. It is followed by several difference frames that include only changes from prior packets, and an occasional base frame. This enables ROHC to survive many packet losses in its highest compression state, as long as the base frames are not lost.

MIDI [21]

The Running Status option of the MIDI (musical instrument digital interface) standard illustrates another way to compress message headers. In MIDI, every message consists of 3 bytes–one status byte which contains the message type (e.g. Note On, Note Off) and two bytes of the data. If the subsequent messages have the same status byte, the status byte is omitted and two-byte messages that contain only the data are sent. The receiver understands implicitly that the status byte is the same as in the previous message.

Related patterns

Data Compression is a more general pattern that accounts for changing the encoding and other ways of modifying the representation of data.

Other patterns referenced in the text

Facade - Design Patterns [14]
Message Channel - Enterprise Integration Patterns [16]
Message Dispatcher - Enterprise Integration Patterns [16]
Publisher-Subscriber - Pattern-Oriented Software Architecture [6]

Acknowledgements

The authors would like to thank their PLoP shepherd, Amir Raveh, for his watchful eye, patience, and constructive feedback.

References

[1] Information Technology–Generic Coding of Moving Pictures and Associated Audio Information: Video, ISO/IEC 13818-2. Technical report, ITU-T, 1995.

[2] Executing Multiple SQL Statements. http://publib.boulder.ibm.com/infocenter/idshelp/v10/index.jsp.

[3] INF: Multiple Active Microsoft SQL Server Statements. http://support.microsoft.com/kb/140896.

[4] Thomas Bayer. SOAP Compression for Apache Axis 1.X. http://www.thomas-bayer.com/axis-soap-compression.htm.

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. Hypertext Transfer Protocol – HTTP/1.0. Technical report, Network Working Group, 1996.

[6] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[7] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory, 2nd Edition*. Wiley-Interscience, 2006.

[8] Phillip Dykstra. Protocol Overhead. http://sd.wareonearth.com/ phil/net/overhead/.

[9] J. Klensin (Editor). Simple Mail Transfer Protocol. Technical report, The Internet Engineering Task Force - IETF, 2001.

[10] C. Bormann et al. RObust Header Compression (ROHC): Framework and four profiles: RTP, UDP, ESP, and uncompressed. Technical report, The Internet Engineering Task Force - IETF, 2001.

[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol HTTP/1.1. Technical report, The Internet Engineering Task Force - IETF, 1999.

[12] Ned Freed. SMTP Service Extension for Command Pipelining. Technical report, The Internet Engineering Task Force - IETF, 2000.

[13] Michael Gallagher and Randall Snyder. *Mobile Communications Networking with ANSI-41*. McGraw Hill, 2002.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Programming*. Addison Wesley, 1994.

[15] Wendi Heinzelman. *Application-Specific Protocol Architectures for Wireless Networks*. PhD Thesis, MIT, 2000.

[16] Gregor Hohpe and Bobby Wolfe. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, 2003.

[17] University of Southern California Information Sciences Institute. Transmission Control Protocol. Technical report, The Internet Engineering Task Force - IETF, 2.

[18] V. Padmanabhan and J. Mogul. Improving HTTP latency. *Computer Networks and ISDN Systems*, 1995.

[19] Tristan Richardson. The RFB Protocol. http://www.realvnc.com/docs/rfbproto.pdf.

[20] C.L. Robinson, L. Caminiti, D. Caveney, and K. Laberteaux. Efficient Coordination and Transmission of Data for Cooperative Vehicular Safety Applications. *VANET'06, September 29, 2006, Los Angeles*, 2006.

[21] Joseph Rothstein. *MIDI: A Comprehensive Introduction*. A-R Editions, Inc., 1992.

[22] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-End Argument in System Design. *ACM Transactions in Computer Systems 2, 4*, pages 277–288, November, 1984.

[23] Dan Winkowski and Mike Cokus. XML Sizing and Compression Study For Military Wireless Data. *XML*, 2002.