Fast-Stacking: Providing High-Performance Stack Compositions in C++

Sebastian Mies
Institute of Telematics, Universität Karlsruhe (TH), 76128 Karlsruhe, Germany
Email: mies@tm.uka.de

Today Internet traffic statistics plot a tremendous increase of bandwidth consumption. New services, like IPTV, Peer-to-Peer, Web 2.0 applications and cloud computing demand for more and more bandwidth. Furthermore real-time applications in networking are converging with non-real-time networks, i.e., IP in cars or Industrial Ethernet. Many of those applications have a common addressing scheme, like IPv4, or IPv6 but differ highly in the used network stacks. This is why Future Internet Projects tend to design a flexible stacking architecture to implement new services, e.g., the 4WARD Project Node Architecture [5]. Those architectures are usually implemented in software and therefore need to be efficient (handle high-bandwidths with low latency) and very flexible (reuse of code for rapid-development) to use them in testbeds.

On the poster, we present a new yet simple interface for network stack composition using C++ templates in a flexible and efficient way that complements those architecture designs. The major contributions are: first, we reduce the glue logic that is usually used in network stacks. This increases efficiency and allows to handle high bandwidths and ensures low latency in experiments. Second, the reuse of layer components equipped with our interface eases rapid-prototyping of new experiments. Furthermore we will use this interface to implement some of our components in the *ariba* software [1] developed within the Spontaneous Virtual Networks (SpoVNet) project.

Light has been shed on the problem defining a suitable interface for communication for many years now. One of the best-known interfaces is the Berkeley Sockets interface. This interface declares basic functionality for a network *socket*. However, it has several drawbacks: First, it is synchronous. This means that threads are waiting for messages to be sent or received and therefore it does not scale very well due to the overhead and limited number of available threads. Second, it copies memory buffers at each stage. To overcome those drawbacks *asynchronous I/O* has been proposed in many operating systems. One well-known portable library that implements this functionality is the *Boost C++ ASIO* [3] library. It implements the *Proactor* [4] design-pattern for asynchronous I/O and uses Scatter/Gather [2] semantics to reduce memory copies.

To enhance this kind of design and to create an interface for network stacks, we must first determine a common subset of functionality we expect from a network stack layer. We decided that the following functionality is essential to build a stack: (1) Connection-free and (2) Connection-oriented asynchronous I/O, (3) Scatter/Gather semantics to reduce copies of data, (4) Flow-control and finally (5) a time-base including deadline timers to implement new protocols. We define this functionality in an interface called *connector*. This interface defines a set of methods to provide the features stated above and is closely related to the *Boost C++ ASIO* Proactor design. A developer can use this interface to build a new interface to a physical network device, as well as for providing new functionality by using another connector implementation (the lower layer) and providing the same to the upper layer. The resulting network stack is flattened at compile-time, due to the use of templates. This means the actual code running on the machine lacks the glue logic and therefore increases efficiency.

We show an example on how to use our interface: First a connector is implemented that wraps the Boost ASIO TCP implementation. In the next step, we implemented another connector to emulate a reliable datagram service. This emulation establishes a new connection when a message is sent to a given IP address and automatically closes the connection after a certain time-out. To circumvent head-of-line (HoL) blocking multiple connections per end-point are established if necessary. Due to stream characteristics of TCP datagrams need a framing procedure. This is done by a framing connector placed between the datagram emulation and the TCP connector. To ensure that connectors support specific features, meta-programming techniques are used to check properties of a connector layer at compile-time. Datagram emulation and framing layer can be reused on any other stream-based connection-oriented connector.

References

[1] R. Bless, C. Hübsch, S. Mies, and O. Waldhorst. The Underlay Abstraction in the Spontaneous Virtual Networks (SpoVNet) Architecture. In *Proc. 4th EuroNGI Conf. on Next Generation Internet Networks (NGI 2008)*, April 2008. CD-ROM.

[2] Douglass R. Cutting, David R. Karger, Jan O. Pedersen, and John W. Tukey. Scatter/gather: a cluster-based approach to browsing large document collections. In *SIGIR '92: Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 318–329, New York, NY, USA, 1992. ACM.

[3] Christopher M. Kohlhoff. Boost ASIO C++ Library. http://www.boost.org/doc/libs/1_39_0/doc/html/boost_asio.html.

[4] Irfan Pyarali, Tim Harrison, Douglas C. Schmidt, and Thomas D. Jordan. Proactor – An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events. Technical report, Washington University, 1997.

[5] Lars Völker, Denis Martin, Ibtissam El Khayat, Christoph Werle, and Martina Zitterbart. A Node Architecture for 1000 Future Networks. In *Proceedings of the International Workshop on the Network of the Future 2009*, Dresden, Germany, June 2009. IEEE.