What belongs in a first networking class for computer science students?

Keith Winstein & Nick McKeown
Stanford University

Abstract

We all struggle to decide what to include in an introductory networking class. As our field has grown, we have been repeatedly faced with the decision as to whether to go broader (to include more topics, like network security and SDN), cut some topics out, and whether to go deep on some essential topics. It is getting harder, not easier, to decide what to put in a one term introductory class. We erred on the side of adding more over time, cramming too much material, leaving students exhausted and probably not clear on what the basic concepts are. We are currently revisiting what belongs in (and out) of our introductory class. This whitepaper is a summary of our current thinking. We would love to get input and ideas from others, and work as a community to develop common practices and shared programming assignments.

Background

Each year we teach CS144 “An Introduction to Computer Networks” to about 120 computer science undergraduates (juniors and seniors), over one quarter lasting 10 weeks of instruction. The original philosophy of the class was to introduce students to concepts and protocols, as well as giving them a hands-on experience of building (in software) portions of the Internet for themselves. The lecture material followed the structure of introductory textbooks, covering some basic properties of networking (e.g., packet switching, layering, encapsulation, routing, congestion control, network security, reliable transport of bits over an unreliable substrate, and the physical layer), alongside several, demanding 2-3 week programming assignments written in C, each taking 20+ hours. For the programming assignments, rather than start from the socket layer and work “up” (building web clients and applications on top), the students worked “down” to build portions of the Internet themselves, in software. In assignments 1 & 2 they built a TCP-compatible transport layer (initially over UDP, allowing students to use shared instructional machines, then migrating to run over a raw IP socket); in assignment 3 they built an IP router in Mininet and in assignment 4 they put everything together to retrieve a web page from a remote web server (e.g., microsoft.com) over their TCP and IP implementations. The class had a reputation for being demanding, and the programming assignments gruelling, but we convinced ourselves it was worth it because at the end of the day, the students had built their own portion of the Internet.

In 2012, Phil Levis and Nick McKeown redesigned the course to use a “flipped classroom”, with 68 short videos (10-15 minutes each) that the students watched at their own pace, with
accompanying quizzes. This liberated the classroom time for collaborative in-class exercises (e.g. using wireshark and mininet), discussions of harder topics and allowed us to bring in guest speakers. While some students preferred studying pre-recorded videos in their own time and at their own pace, the class earned a reputation for being even more work than before. By the end of ten weeks, the students seemed exhausted and it wasn’t clear they walked away with a clear understanding of the concepts we wanted them to know. It seemed time to re-evaluate the class once more, to rethink how best to introduce the concepts of networking to our undergraduates, and prepare them for a career in networking, whether they are developing portions of the Internet themselves (e.g. at a cloud provider), writing applications to exploit it, or as networking researchers.

We are in the midst of re-evaluating the class. While we have conducted some small experiments, we have not reached firm conclusions. Rather than propose a new approach in this whitepaper, we attempt to document the challenge, some guiding principles and our current draft plans. In the spirit of the workshop, we offer these ideas for discussion, and welcome ideas and input from others.

Networking in general, and the Internet in particular, are central to computer science today; it is very important to prepare our students to use it and improve it. We would welcome working together, as a community, to figure out what to include in an introductory class, and to share the development of meaningful programming exercises.

What to include

Over time it is becoming harder, not easier, to decide what to include in a one term introductory networking class. For example, while we would like students to know basic concepts of networking, do we only focus on the Internet? Is there room, or is it relevant, to spend time on alternative networks any more? We would like our computer science students to roll up their sleeves and write programs, but it is hard to define manageable, time-efficient, educational assignments for functions that typically live in the kernel and are written in C. And given that an increasing number of network owners are rolling their own code to control their network (e.g. cloud providers and ISPs), should we focus on legacy protocols, or instead open our students’ horizons to the possibilities of defining new network behaviors for themselves? Put another way, our field has not only grown, it is also in transition from being defined by fixed protocols to encompassing entirely new, non-standard behaviors inside large homogenous data-centers. How can we pack the old and new concepts into a one term class, and provide our students with hands-on programming experience, without overwhelming them?

---

1 The class was also offered as a self-paced MOOC for anyone to take for free, but is not our topic here.
Guiding principles

Overall, we want our students to take away from the class a clear understanding of the basic principles of networking (so they can study more if they wish), and an introduction to writing network programs for themselves (allowing them to write more in future); while not burdening them with too much details and minutiae of specific standards and protocols.

- There are some basic concepts every student should thoroughly understand: layering, encapsulation, packet-switching, how to send data reliably over an unreliable network.
- There are some large topics they should study, to lay the groundwork for them to study more if they need to in future: routing, congestion control, the physical layer, controlling the rate and delay of packet flows.
- We want students to become comfortable writing programs that implement portions of the Internet, that are capable of interoperating with each other and/or with existing software and equipment.
- We want to expose students to basic measurement tools (e.g. wireshark, traceroute), and emulators (e.g. mininet)
- Aspirationally, we want to give students experience programming the control plane as well as the data plane; to potentially think of a network (or at least some homogeneous portions of the network) as a distributed system, that they can program, top-down, with more focus on distributed system properties (consistent state, distributed decision making, remote execution) than on protocols.
- In 10 weeks, without burning them out.

Emphasis on Conceptual Learning… With Partial Success

Our class is a work-in-progress, and not everything has worked out well each time we have taught it! We have chosen to emphasize:

- “learning by building” (lab assignments ask students to implement significant portions of the Internet, e.g. TCP, an IP router, network interfaces that perform ARP), including practical debugging skills like understanding wireshark output. The week-by-week design of these implementation assignments is listed below (please see appendix), and we are happy to share these assignments with instructors of other networking classes.
- abstractions that recur throughout networking, e.g., reliable byte streams, buffers and queueing, remote transactions in the presence of unreliability.
- reasoning about the behavior of a distributed system.
- notions of reliability, consistency, and fairness.
- why the Internet protocols behave as they do. For example, we try to make an explicit connection between the maladies that may afflict an Internet datagram (e.g., corruption, loss, duplication, reordering) and the corresponding mechanisms that TCP uses to transform datagrams into a reliable byte stream ( checksums, retransmission triggered by timeout or acknowledgment of later-sent segments, sequence numbers, long wait by one
peer at the end of the connection). Similarly, we try to teach why the AIMD algorithms for congestion control are designed as they are and what notions of fairness they can (and can't) provide. Ultimately we feel good when students are able to answer questions about how the need for these mechanisms might change on different underlying networks.

By contrast, we have chosen to de-emphasize some concepts that appear in other networking classes, including

- **Distributed routing protocols (RIP, OSPF).** We still teach these, but we don't ask students to implement them or spend as much time on them as perhaps a traditional networking class would. (Stanford classes are ten weeks long.)
- **Protocol header formats.**
- **the OSI model.** We have found that students taught first with the OSI model are sometimes then confused by "stacks" that connect modules in a nontraditional way. For example, TLS transforms one reliable byte stream into another, with the addition of cryptographic security. Any protocol (e.g. HTTP) that can run on top of a reliable byte stream (e.g. HTTP-over-TCP) can also have TLS inserted into the mix (HTTP-over-TLS-over-TCP, aka HTTPS). But students who are too wedded to the OSI model as a prescriptive, rather than descriptive, model are sometimes confused by this--"so what's at which layer now"?
- **TCP's three-way handshake** and similar protocol details. We believe the SYN/SYN-ACK/ACK exchange can be more cleanly taught as a consequence of fundamental or conceptual rules of TCP: (1) “to indicate the initial sequence number, every outgoing stream begins with a SYN flag”, (2) “every TCP segment that occupies a sequence number must be eventually acknowledged by the receiver”, and (3) “it’s helpful to piggyback flags or data in a single segment if possible!”
- **“TCP vs. UDP”**: We suspect that the OSI and Internet stack models lead students to give outsized weight to UDP as an “alternative” to TCP, which we think may be a conceptual error. We would rather teach students to see UDP in terms of its function: as a translator between two similar services--Internet datagrams and user datagrams--and better regarded a thin shim on top of IP that other protocols may be built on top of (e.g. QUIC, BitTorrent/uTP, DNS with its simple retransmission scheme).
- **Network security.** MACs, public keys, certificates. Everybody should know this, but we have other classes in our department that teach this better than we do, and don’t think we have time to give this subject more than a cursory overview in our 10-week class. We might remove it completely.

This style hasn’t always connected well with everybody. Students sometimes have had trouble connecting what they hear in the class with what they have read or learned elsewhere. After we spent most of a quarter emphasizing the descriptive, rather than prescriptive, nature of the OSI model, many of our students were mystified when we had a guest speaker from industry who kept talking about the difference between performing some functions at “L2” instead of “L3”.

In general, though, we find that computer science student’s typically don’t struggle the most with issues of layering and encapsulation, or with routing algorithms, or with concepts related to convergence of a distributed system. For students with a CS background, the most challenging areas of the class have typically involved grappling with packet switching and its vagaries: congestion control, de-jitter/playback buffers in video and audio streaming, and the statistics of packet timing or the detection of loss. We increasingly believe that every student should gain a deep understanding of the consequences of packet switching and how to deal with its unpredictable behavior. This, we believe, is a fundamental concept of utmost importance, and it is worth going deep and spending significant time on it. Armed with a thorough understanding of packet switching, students are better equipped to understand many important aspects of networking, including congestion control, video streaming, video-conferencing, rate and delay guarantees, fairness and low-latency communications.

In our rush to include more and more in a computer networking class, it’s easy to end up giving a more watered-down treatment of these concepts. But we don’t feel right about that either—they are the areas closest to the core of networking.

Where we are headed

- **Programming Assignments**: Students build the core Internet building blocks (TCP over IP, IP network interface, IP router) in a modular ground-up fashion. (see below)
- **Class time & Lectures**: Focus more on principles, less on details and standards. For example, when teaching congestion control, focus on approaches and principles than, say, on specific versions of TCP. What are the detailed packet dynamics and consequences of AIMD, rather than on the details of the specific versions.
- **Over time would like to add:**
  - Hands-on experience experimenting with using/measuring the network.
  - Hands-on experience with how a network is controlled (if we can figure out how to include this).
  - Basic principles of distributed systems.
  - Students building pieces, then connecting together with other students to form a network.
  - Clearer way to teach congestion control
    - What is it? Is it a set of mechanisms to avoid collapse? To achieve fairness of agents with limited visibility of one another? To achieve good performance?
    - Is it a niche topic in networking that academics are unreasonably fascinated by, or does it encompass any question of resource allocation?
    - Is it an intractable problem: multi-agent algorithms for resource allocation in the presence of partially observable current and future constraints, goals, and behavior?
    - Can at least we nail a cogent understanding in the students’ minds to this question: *If AIMD is the answer, what is the question?*
Proposed discussion questions

In the interest of stimulating discussion and debate at the Workshop, here are a few questions to consider:

- As a community, how can we (or should we) work together to create a common set of programming assignments? They take a lot of work to create - should we pool our efforts? If so, how to keep them evolving? What about solutions and plagiarism by students?
- Should we go broad and teach everything lightly, or go deep on some topics? If so, which topics to go deep on?
- How important (in a first class) are things like congestion control, or fairness, or the basics of packet switching and its statistical consequences?
- Most assignments (including ours) focus on the reliable dataplane between applications. What about the control plane? Should we expose students to writing their own routing protocols, for example in an “SDN-like” (e.g. ONOS) framework?

Appendix: Our Implementation Assignments

What we tried: Building the core Internet building blocks (TCP over IP, IP network interface, IP router) in a modular ground-up fashion.

1. Week 1: **Intro sockets programming and reliable byte stream.**
   - Demystify Internet protocols. Teach students to fetch a web page by hand (with telnet) and send themselves an email message from an arbitrary address (typing SMTP with telnet). Students are often amazed that it is this easy!
   - Students implement a (very) simple HTTP client in C++. This uses socket/connect/read/write/close, but with a support library to ease the mechanics. It is no more than 10 lines of clean code.
   - **Then**: Create a module that exposes the abstraction of a buffered reliable byte stream with limited internal storage. Explain that the goal of TCP is to convey this abstraction over an unreliable datagram network.

2. Week 2: **Stream reassembler.** Create a tool that receives substrings (taken from the same string at any absolute offset and length) and reassembles the byte stream, while using limited internal storage. (This is effectively a receiver for a sliding-window protocol, but has nothing specific to TCP.)
3. **Week 3: TCP receiver.** Create a receiver for the TCP protocol, that receives TCP segments and converts them into a reliable byte stream (and calculates the cumulative acknowledgment and window size for feedback to the sender).

4. **Week 4: TCP sender.** Create a sender for the TCP protocol, that accepts an outgoing byte stream (from the application) and an ack and window size (from the receiver), and converts the outgoing byte stream into a series of outgoing segments.

5. **Week 5: TCP socket.** Wire up a full implementation of TCP, connecting the TCP sender and TCP receiver. Phenomena like the three-way handshake become artifacts of the basic principles of TCP, rather than first-class considerations. The student’s TCP socket exposes the same interface as the kernel’s. **Result:** by changing the name of the socket used, the student’s HTTP client from Week 1 can now fetch Web pages using the student’s implementation.

6. **Week 6: IP network interface.** Create a network interface abstraction that accepts a series of outgoing IP datagrams and converts them to a series of outgoing Ethernet frames. Must implement ARP to learn destination Ethernet address. **Result:** the student has a network stack from TCP down to Ethernet.

7. **Week 7: IP router.** Given a routing table, and an IP datagram, which network interface should it be forwarded on? The goal of this modular structure is to make sure students are not thinking about ARP at the same time they’re thinking about routing.