A DESCRIPTION OF *CONTENT-CENTRIC NETWORKING* (CCN)
--based on a Special Invited Plenary Short Course by Van Jacobson at the Future Internet Summer School, Bremen, Germany on July 22, 2009

1. **The Goals of CCN: Simple, Universal & Flexible**

Content-centric Networking (CCN) has ambitious goals, an entirely new communication architecture that works as well as the Internet, at least as well as TCP/IP in every dimension. The objective is not to replace the Internet, but to evolve from it, to take another step. CCN's motivation: the Internet changed the world, but it was designed for the world of an earlier time. Now, we can see the cracks; it doesn't fit the world that we have today.

There's real pain in the current Internet, particularly in Security, which has to be far more fundamental. Just hooking up the current Internet – getting things to work at the Internet level and at the application level that runs over it – that's gotten to be very difficult, much more difficult than it should be. There is a model conflict; the problems that need to be solved and the ways that are available to solve them no longer relate. And because of that, there needs to be a translation layer.

Looking at programs from five or six years ago, we are finding – increasingly – that they were middleware. You'd have an application layer that was fairly slim; you'd have a network level that was sockets, and it was fairly slim. And you'd have a huge chunk of goop in the middle that was translating between the applications model of the world, which was generally content-focused, and the networks model of the world, which was host-focused. And that's a very hard translation. They are fundamentally different.

A network that works more in the applications world would allow the middleware to go away. You wouldn't have to translate between models because they'd be the same model. In effect, a universal middleware would be developing. A networking infrastructure would be developing that's designed for communication, works in the applications content model, and overlays on top of IP. The translation would have to be done once, in one way. This would be a nice evolutionary story, sort of the same story as the Internet's.

The original Internet rollout was four nodes: UCLA, Salt Lake, SRI, and Berkeley. The communications protocols running between those nodes were demonstrated; they could talk to each other, and they could talk to other computers on those campuses. The virtue of this network was that it concatenated networks. That's why Vint Cerf originally named the Internet "Catenet." It stood for "concatenated networks." It was designed to be an overlay that abstracted out the behavior of the lower-level networks, these things that today we call "layer 2" – in a uniform way to allow you to glue the pieces together and provide transitivity. That meant networks could grow without having to translate between the different protocols, because the translation was done once in the IP layer, in the shim layer.

At a time of a lot of evolving technologies, like long-area modem links were slowly getting faster, from 9600 baud per second to 56K to 128K to T1. The newly-invented Ethernet provided 3 megabit per second connectivity and then evolved to 10. And then there were lots of experimental wireless links that were being used for digital and military communications. Those were all networks that worked in isolation, but we wanted them to work together, and we wanted them to work together without having to do a lot of human configuration work. And the Internet gave us that.

When UCLA bought a bunch of computers and plopped them down in the computer room, they wanted to hook them up to their Internet test infrastructure. They put TCP/IP on them, and they built an Ethernet driver. And then suddenly, all those computers could not only talk to one another, but they could also talk to the computers at Berkeley and SRI, because these dissimilar networks concatenated; they joined together. And that was done by the machines. It was the objective of the protocols to provide that concatenation, so people didn't have to think about them. And that was completely revolutionary, because all networks up to that time had to be manually hooked up, manually configured to make them work. And to have a network that hooked up itself was tremendous value. Many more people could start to do networking. That caused the Internet to grow inside of campuses. The machines were bought, and the Internet hooked them up – little viral clouds to the Internet, growing in lots of different institutions. And then the NSF got the bright idea of putting a wire between those institutions, and suddenly a big, national network was born. And at every step, it presented some value.

Our hope is that future Internet technology, content-focused technology, could grow in the same viral way – with little, limited deployment, and providing value, maybe even value as middleware. Applications would work better, more seamlessly, and while requiring less configuration. Over time, as the islands grow and start to concatenate together, pretty soon there could be an entire network that works in a content-focused way. And as the lower layers could be ignored, they would start to wither and go away.

2. **CCN can run over anything, and anything can run over CCN.**

A communications architecture that works over anything is automatically an overlay. Because of that, "overlay" is not necessarily a useful term. But the "expressibility" of the thing is a useful distinction. IP started as a pure overlay on the phone network, so people did not think of IP as a network. The phone system was what provided the connectivity, the networking. Some thought IP was just a waste of a bunch of bytes with headers.

Starting in 2003, things flipped. The amount of phone traffic that was carried in packets had steadily grown. By that time, 80% of the world's phone traffic was carried in packets, in IP packet over SONET, because it scaled. And the phone service had turned into what was primarily an edge service: you packetize, and then you distribute via IP. We went from IP overlaid on phones to phone systems overlaid in IP. It took about forty years, but it showed something about the expressibility of the IP network: Not only can it be overlaid on anything, but almost anything can be overlaid on it.

The intent is for CCN to have the same character, but more so, a truly universal overlay. It's more expressive than IP. It's easier to overlay it on top of other things because it puts much less demand on the lower layers. IP really requires a point-to-point instruction, a lower layer. And it dislikes broadcast links and links that don't have addresses. CCN loves them because it doesn't talk to things; it talks about things. It can be overlaid in pretty much any communication link available. Because it can be overlaid on IP, it can be deployed now as middleware. IP can also be embedded in CCN. IP can run quite easily over CCN. Only a little bit of space is wasted because the CCN headers are bigger, and the functionality is as complete.

3. **Why CCN (and why it's so difficult to explain)?**

To understand it, you need two different ideas in your head. Having only one of the two, it looks pretty nonsensical. One is a different model of forwarding, a different model of the moving parts of the network, because it's a network that doesn't talk to things. There is no host abstraction. There's only a content abstraction. And that makes for some surprising differences in the forwarding machinery. It is as simple as the IP moving parts, but they're different moving parts. They apply to a particular content model.

When you read RC791, the description of IP, you may say, "This moves packets around, but how is the job really done? I've got this file, and I want to move it from here to there. How do I do that?" The answer is not in 791. It's a transport problem.

Read about TCP/IP, and then it's all pretty clear. Without reading TCP, a lot of the things done in IP seem silly or pointless, particularly coming from a background of, say, X25, which is a different model of what the underlying network is. Beyond silly and pointless, they seem like they're not going to work, because IP doesn't provide the basic capabilities needed for that model of transport.

CCN presents the same issues. The transport model must be understood. But unlike TCP, where you're trying to move a byte stream, in CCN, you're trying to move a fairly rich object space. You cannot understand what you're trying to accomplish with the moving parts until you understand the content model. Unfortunately, it's difficult to describe two things at once.

This description begins with the content model and proceeds with the moving parts. Then, perhaps, it's easier to see how the content model meshes with the machinery. From that point, it explains how routing is accomplished when you've only got content and how transport is accomplished when you've only got content – with a few examples and some measurements from the prototype implementation.

4. **What the world looked l like (when the Internet was invented)…**

It was a very different world. There were only a few computers. They generally weren't connected to anything. And if they were connected to anything, it was one connection. Everything about these computers was expensive. They lived in little glass enclosed rooms, and they had a whole cadre of priests that attended to their every need. Normal users couldn't go in and deal with a computer. Students who wanted to run a program, they'd hand in their tapes and a card deck to an operator, who would take them in and feed them to a computer, and, hopefully, give them back their program and tapes.

It was a world where what was important was keeping the machines busy because they were so expensive, and having the resources available to these expensive machines so that they could do their job. And a lot of resources were needed: card readers, punches, tape drives, printers. And all of them were very fabulously expensive by today's standards. A printer cost as much as a house. The main interest was sharing these resources. With a communication link, one could say, "Maybe I don't have to buy two printers. Maybe I can buy one, and use it on two computers."

This is why the networking protocols came into existence – so that resources could be shared between computers. The goal was not about sharing data. In those days, data didn't live on computers, but on tapes and in card decks. It was carried around. Networking created today's world of content, but networking was not designed for it.

Today, networking does not concern itself with sharing resources. Computers are disposable, ubiquitous commodities. My laptop has more than a dozen computers in it. Nothing about them is particularly expensive. The economies of manufacturing have gotten so large. How this happened was largely through networking. Computers aren't particularly important, but the data that's on them pretty much defines our lives.

But when the Internet was invented, because of its goals – to share what were then expensive computing resources – the network architecture's central abstraction was the computer, this thing people no longer care about. And because the objective was to extend (share) computing resources, what those abstractions did was have a conversation, e.g., let me talk to the I/O bus. The intent was to talk to a printer, to a tape drive, and over a network connection – primarily point-to-point conversations between exactly two hosts.

5. **Consequences Today**

[This original concept of] Networking hates everything about the world today. It doesn't like mobility. It doesn't like things to move around because its model is two fixed points that are having a conversation. It doesn't like intermittent connectivity, another instance of things moving around. Since there's so much content in the world, as applications are being written, there's the mindset – what information is needed, what information must be presented? It's a world of content abstractions. When somebody actually needs to move information around, they have to change that into host abstractions, and they ask, "Who do I talk to in order to get the data I want?"

After you figure out what your app is supposed to do, there's another question – where is the server? You have to bind the "who" in order to make the "what" work. And that's done largely through configuration.

The apps really just care about what. The abstractions have to be changed. There is no relationship between them. Where the data is cannot be algorithmically derived from what the data is. Some manual configuration or some configuration infrastructure is needed in order to tie things together, which has led to this huge growth of middleware and non-functioning networks, because they're so easy to misconfigure.

Lastly, this value of this resource-sharing network was to provide a bit pipe: to move bits that were completely opaque to the network from one machine to another. And to the extent that the data could be kept completely opaque, the more devices the network could support, the bigger the range of conversations, and that was goodness.

What it means, however, is that the network has no visibility into those bits at all. It has no concept of what is being done over that wire. And as it relates to security at the network level, what it can do is armor the wire, because that is all it knows about. It can ensure that the bits in transit don't get corrupted. But I don't care that the bits in transit didn't get corrupted; I care that the bits received are the ones I wanted, from the person they purport to be from (for example, from the intended bank and not from some scammer), and that there was nothing added or deleted. The network cannot guarantee that. It can't help with that problem; it can't see the content. And yet there is a lock on browsers that says, "SSL, this connection is secured." But it is not. The security in human terms is completely invisible to the network. And that's a deliberate result of the fact that the content is opaque.

Today, all we do with the Net today is move around content. According to Andrew Odlyzko's (UMN) MINT data, by the end of 2008, the Internet was moving 8 Exabytes a month[i], which equates to 25 Terabytes per second, every second of every day. The theoretical capacity of a fiber has finally been reached. It took us a long time to get there.

And all that data is not just shuffling around stuff that already exists. It's being produced at a prodigious rate, particularly since all of our new devices can automatically upload video to YouTube. There is a huge content creation explosion. A company called IDC did a study in 2008, where they found in 2006 alone, 180 Exabytes of content were created[ii]. Most of that, 8 Exabytes a month, is new stuff. They have done the study every year for about a decade, they found 60% annual growth, projecting more than a Zettabyte being created by 2010. A Zettabyte is 10 to the 21st; there aren't enough letters remaining in the alphabet for the size of content we're creating; it is unimaginable amounts of data, trillions of Terabytes of data.

6. **Networking and the Cost of Storage[iii]**

The proliferation of data is being driven by the evolution of the cost of storage. Thompson from IBM did a survey article IBM R&D on the cost of IBM disks; I graphed ten years of that data, showing an exponential decrease.

Another set of data comes from Doug Galby, chief economist at the FCC, was surveying carrier prices, I picked some OC3 prices, dollars per megabit per mile. So storage has seen an exponential decrease in cost; networking has been flat (or at least not exponential; data is harder to come by).

In 1990, the United States was making a transition at the customer level from copper TDM infrastructure to optical infrastructure. Internally, the network had largely changed to optical, but the tails going out to customers – what could be bought -- was all copper. The fastest available was a T3. And then suddenly, around 1990, faster options were obtainable – these 150 Megabyte OC-3s on fiber. It was sold at a relatively high price, but the actual price per byte was going down, e.g., "Get 3 times the bandwidth at only twice the price." People began to dump their old T3s and put in OC-3s, so they could retire the inefficient, expensive copper infrastructure.

By 2000, internal backbones had all been upgraded to OC-48, and were moving to OC-192. And multiplexing is really expensive. The less muxing needed for a customer, the cheaper costs are internally, so there was a significant advantage for the telcos to push OC-12s out to the customers, rather than a bunch of OC-3s. Because they had a much fatter backbone, they were offering higher speed links at the edge to customers. And it was the same deal, e.g., "4 times the bandwidth and only twice the price." Costs to customers would increase by a factor of two, but telcos' costs would go down by a lot more, because an OC-3 actually costs more than an OC-12. The telco business model consists of a lot of inertia. There's just no way – given technical constraints, financial constraints, regulatory constraints, etc. – that they could drive their costs the way that the disk could drive its costs.

But disk costs – that decrease is staggering. It's been falling 3% per week every week for the last 25 years. And recently, with new recording technologies, it's been falling even faster. In the future, that curve is likely to accelerate, because the one thing that nanotech does really well is produce large regular structures. Within the last couple years, the wonderful things to come out of labs are all big regular structures that function as memories.

There are many examples. There's a new self-assembling nano-capacitor array[iv] – nanotech fabrication – that uses the PZT ceramic used to coat pots. It's dirt cheap, and the first cut out of the gate, minimum density: 200 Gigabytes per second. That's the density on a platter in a lab. It is state of the art distensity, way beyond anything on the market today. There's a new MEMS array[v] – standard silicon fabrication techniques – storing bits in a mechanical structure, flipping little gates around: 4 Terabits per square inch, 50 times the best you can do with disks, requiring no power to run, and completely non-volatile. In February, LBL announced this new self-assembling copolymer structure[vi], 10 Terabytes per square inch. They also announced this new really cool thing that embeds magnetic particles inside carbon nanotubes[vii]. A field is then applied to move the particle to one end or the other of the tube. It's only a Terabyte per square inch, but the particle is completely sealed in its own little environment in this tube, so there's no way to disturb the state of the bit. There's a per-bit lifetime of trillion years – the first electronic technology that has a longer lifetime than stone-chiseled tablets.

All of these things are happening, they're all really cheap, and they're all storage. The demand is there because of the content creation cycle, and the technology is there to produce these very fine structures, being scaled down to the atomic level with densities just shooting up. Given these trends – the bandwidth trend, which just can't evolve that quickly, and the storage trend, evolving like lightning – it's best to trade storage for bandwidth. Whenever bytes are pulled across a wire, it's going to be more expensive. The problem is storage and communication work in completely different worlds. The storage world is what you want.

The way to get a file is to present a name: the bits associated with that file are given back. The way to get a file over the network is to find a server, then ask it to give the file. So in a communication problem, the primary issue is who you talk to. In a storage problem, it's what you want. And that's the model conflict; these two don't substitute very well.

The important thing to realize is that the original Internet abstraction wasn't God-given. It was the right thing to do at the time -- in the 1960s. Now it's not the 1960s, so the question is: can a network have the properties of the Internet – same scalability, performance, and robust operation – but with a different abstraction? And can the abstraction be a content abstraction?

7. Self-moving, Self-managing Content (and its context)

A model of how to accomplish the goals (above) at a 50 thousand-foot level can be described with the example of calendar data, which should be able to move around all by itself and manage itself. It gets updated on phones; it gets updated on laptops; it gets updated on home machines; it gets updated on work machines. And people with access can put, view and change items on somebody else's calendar. Because it's sort of a distributed set of data, there's no single object in one's life that has a consistent view. It would be best if it ran itself.

In this model, a person enters their office with their phone, which broadcasts interest in the person's calendar to all of its interfaces. e.g., Bluetooth, 802.11. The phone asks, "Anything go on the calendar? I'm interested in the calendar."The desktop hears it and says, "I have something for the calendar. This person has a meeting." And it supplies that information to the phone. The basic model is that devices declare at a high level what data interests them, and other devices hear those Interests, compare them to the data they (the devices) have stored and respond with data matching any received Interests.

Then the person leaves work and goes home. The laptop at home plays the same game. It says, "Anything in the calendar?" The phone says, "There was a meeting today." All of those devices end up in sync. But the person did not tell their phone about the desktop. They didn't say, "Query the desktop to see if it's got new calendar entries; query the laptop to see if it's got new calendar entries." This is an n-squared problem. All of the devices have to be introduced and taught (configured) to do the plumbing.

It is possible for the user to work at a much higher level, to tell the device to keep track of the calendar and to manage the details. In this model, the person is the network, the one that moved the calendar entry from work to home and moved it into the memory of the phone. It doesn't matter what the communication vehicle is. If it moves bits, it's a network.

The person specifies the objective, not how it's accomplished. The devices deal with how to accomplish it. The data appears wherever there's interest in it, so it manages itself. It goes where it's wanted, not anywhere else. It's a model that loves broadcast, because it's not talking to things, it's talking about things. It's asking questions, e.g., "Do you have the calendar?" It's like asking if anyone has the time. A person doesn't need to be introduced to everybody in a room to ask that question. It can be asked of a room. And a good answer can be provided.

CCN works at that same level. It works at the data names, not at the participants' names. It asks by name, so the difference between communicating to storage and communicating over a wire goes away. They are treated exactly the same way. They are the same abstraction at the networking level, at the communication level. Therefore, data integrity and security become even more important.

At present, there is the notion that security can be guaranteed by armoring a wire and making a trusted perimeter around the machines. But this is false, of course; it can't be done. People just pretend it is true. It does work sometimes; sometimes, it slows down the attackers a bit. But as it relates to wireless communications, it is just a fantasy that is not supported at all, because nobody knows what is talking to them. It is wireless. There's no connection; the source of the RF could be anything. This was the first lesson telcos learned when they went from landlines to mobiles. With a landline, they knew exactly where to send the bill – to the home attached to the wires. A location was defined by the physical attachment to the network. What's attached to a cell phone? Nothing. Where does the bill go? There is no physical connection that can be used as some source of identity or some identification of the service. Only the data.

So right after the cell phone was invented, the SIM chip was invented. Which was necessary to provide some cryptographic assurance of the identity of the subscriber. Since it could not be acquired out of the wires, it had to be acquired out of the data, because there was nothing else. And this made security vital, fundamental to any service. And that's a segue way to the content model.

If we're going to be facing zettabytes of data, information better become self-managing; nothing else scales. Nobody lives long enough to deal with that much data. The sun won't live long enough to deal with that much data, if it's going to be handled byte by byte. It has to become mostly automatic, mostly run by the devices themselves. And in order for data to self-manage, e.g., delegating the calendar sync, then the things you delegate to need context, or they won't work. There are three types of context.

First, ontological context: The correct information has to be identified. For example, with the Calendar sync, what spans the calendar must be known, and unwanted (non-calendar) stuff

must be prevented from being synced. In this case, HD videos wouldn't need to be moved to the cell phone. And the correct data should not flow to the wrong place.

Second, provenance of the information and the specific communication: What matters is not who is supplying the information, but what the information is, yet who created the information needs to be within the information itself. For example, just because something attests that it is an item for the calendar, it is not necessarily true. There are a lot of spammers in the world. And there a lot of non-calendar scams that could generate actual revenue, like forging an charge on a bank account. There must be some basis for trusting the information, and it has to be in the information itself.

Third, location: What is done and attempted depends on location. The original Internet design deliberately didn't include locality. It was not possible to know if communication is with something adjacent or something across the world – by design, which was a good design at the time. But if somebody is attempting to move videos from a camera to a media server, they do not try without a viable connection, a way to talk to their home video server. It would be too expensive just to try. It would better to know local context before doing anything, since it affects what should be done.

8. **Friction**

There's another set of issues. By trying to automate things, to get things done on a user's behalf and work at a higher level, a fairly simple action could result in many complicated actions, e.g., amplifiers. Amplifiers can amplify bad as well as good, and that needs to be engineered out to the extent that it can, which is difficult. It's easy to make mistakes, particularly when data is moving around; we don't want to amplify mistakes.

For example, a really popular file-sharing device, FolderShare, presents the abstraction of a single file folder that exists on an arbitrary number of machines. If a file in FolderShare is deleted accidentally, and it's one of those "oops" moments, FolderShare doesn't know that. When a file is deleted, it makes sure it gets deleted everywhere, so the backup vanishes as well. Nobody wants to amplify that mistake, a really common mistake. In today's world, there are unfortunately a lot of bad guys who realize there's a lot of money floating over the Internet, and they want that money.

There are many ways of attacking us from the Net and by the Net. When we're amplifying our actions, we don't want to amplify their actions – the attacks. Right now, that's possible for the attackers, because of the packet forwarding model – the point to point model with transparent data flows – is what's called a gravity model. It looks like a sink. There are a bunch of pipes connected to it, and the data has to run down. The data is not recognizable; it is opaque. It's important to verify that all the pipes go in a downward direction and none of them have any loops. It's a topology maintenance problem, and why routing protocols are run: to create and distribute data over a sink tree. The root node of that tree is the destination of the data. There's one path from any source to that destination.

Another way of looking at it: The sync tree is a lens that attackers can use to focus traffic on the destination. This is the basis for DDoS attacks. A zillion zombies can be recruited from Microsoft's insecure operating system. A little program sends a little bit of traffic, but if it's from a hundred thousand machines, anything can be blown out of the water. Google can be blown out

of the water. And because the lens only works one way – it's asymmetric – all the advantage is on the attacker's side. It's difficult to remediate that attack, to push back towards the attacker. The Net is just doing its job, focusing traffic, it's collecting it. It's delivering it to the destination.

This was an engineering choice that was a consequence of other engineering choices. The lens has to be made. It comes from the data transparency in the protocol. Maybe we don't have to do that if we use a new model, In particular, our thesis is that a content model would make it more difficult for attackers and easier for defenders.

9. **Getting Rid of Useless Abstraction**

If the goal is to self-manage information, the three types of context are needed, and friction is not. For this to be accomplished, first, we jettison the host extraction; it gets in the way and provides no value. The way to get ontology is by a hierarchy. Meaning comes by organizing, knowledge in a structure, hierarchical structures. And they're represented as hierarchical names.

Provenance comes by signing, which is the cryptographic primitive for giving provenance, i.e., establishing identity or asserting identity. And it's a particular type of signing – not of the name, not of the data, but of the binding between them. The binding is what is important. To understand why, the last eight years in the United Stated serves as a good example. During this period, the country learned a lot about lying. There are two styles.

There's a Cheney/Bush lie, where they just made something up and said it, i.e., just corrupt the data. But if the data are signed, if it's self-certifying data, that kind of lie can be detected. Another style is a Rumsfeld lie, where the answer is true, but it is not to the question that was asked. Somebody asks, "Are there really Weapons of Mass Destruction?" And the answer: "Yes, I had oatmeal for breakfast." This second style of is harder to detect; it would self-certify, but it's totally irrelevant. The only way to establish relevance is by looking at meaning and context: the ontology. It's the name that provides the context. So to prevent this kind of an attack – which is very common because it works so well – you need to certify bindings, not just the data, not just the names. That's how to generate provenance, by signing everything. To get locality, the way that bits flow must be made visible at the network layer. Bits diffuse from a producer to a consumer. That's the way network works; they can't be teleported. They have to move. Since the early days of telephony, everybody has tried to pretend this isn't the case. While talking to a telephone, people generally view themselves as actually talking to the person on the other end. But they are not. They are talking to a phone. There are some complicated electrical things that are happening, and the conversation is diffusing through a network, and it's being reproduced on the other end. The person is actually talking to something local, and when they're communicating, they're communicating to something local. If we make that explicit, not hidden, then we can use that locality as part of the engineering. That awareness can be part of the protocols and exposed higher up in the stack. There is no attempt to extract that out; instead use a diffusion model. The way to get rid of friction is to correct a mistake that John von Neumann made, which is thinking of information in terms of its containers.

10. **The Right Bits and No Containers**

Security on the Internet is really bad. It just totally sucks. And that's not because of sloppy programming, though there is a lot of sloppy programming. It's far more fundamental

than that. The way we view information is not the information itself, which cannot be named, but we view it by where it lives, i.e., its container, which can be named.

When you get a webpage, you get it from a host, a container for a bunch of webpages. You get it from a file on that host. In your mind, that's what you're referencing, and you hope that the bits in that container are what they're supposed to be. But you have no insurance of that. And they – a bank sharing your account information, for example – have no insurance of that. It's difficult to know what's in the container. But it's the basis of computing, going back to the early days of computing, in the 60s and even earlier in the 50s, in the world that was described earlier, when machines were expensive and memories were tiny. Back when disks were kilobytes. For example, a serial machine in the early 60s may have had 4 kilobytes of serial delay line memory, and that would have been a lot. It may have had a Fortran compiler on it, in addition to the normal utilities, which could have been done when there was a whole 4 KB of memory. It wouldn't have had a disk. It would have had a paper tape reader, which would have probably accommodated 12 KB on a spool of tape. The size of memories at the time was the reason why data didn't live on machines. When you wrote programs you viewed the data as coming from some external source which was abstracted – the tape drive or staging the data into a file on the disk. The author would read from input, which was a container where the data was dumped, and would write to output, which was another container where you put the results were placed. You didn't deal with the data directly, so you didn't name it, because you had to keep replacing it due to the limited storage, right down to the machine level. A lot of high-speed memory was not available. Machines had registers for eight locations of high-speed storage, and computations were done in the registers. And RAM was much slower.

This now well-established mindset of naming the containers to operate on their contents is totally insecure. It's the source of all bugs[viii]. It is not possible to know what is in the container, even if you track its the history. And if there's any sort of parallel processing happening, something can override it. And this relates to security properties. There is, for example, the fallacy of the un-pickable lock[ix]. A lock, by its nature, has to have the function of being opened. If it can't be opened, it would be useless. Welding something shut would be more useful. But if a lock can be opened, it can be opened by both good guys and bad guys[x] And a container, by its nature, you need to be able to replace its contents, or it is not fulfilling its mission. But bad guys can replace the contents, too. *All the currently extant attacks on the Internet have to do with attacking containers.* Common attacks, like cross-site scripting, make a container command a webpage to point somewhere other than where the user thinks, but you can attack many other kinds of containers. You can attack the host container by attacking ARP to get the MAC layer to IP address binding, or by attacking routing in order to divert the path binding to divert the data elsewhere, or attack DNS by attacking the name-to-address binding. All of these are levels of indirections, ways of building containers, moving you away from the data; they are easy for attackers to circumvent.

To really secure data, you must secure the data, not the thing that holds it. These containers were also not God-given. To develop any sort of code, you probably use some version control system, svn, cvs, git. There, you don't want a file container, because you might want to back up and start over. If you had a container, your history would be gone. A version control system keeps the temporal history. You picture yourself working on files – that's what the editor sees – but in reality, you're working on this tree-structured set of information that captures all the

versions. They can go back in time, which is something everybody frequently needs to do. It's a model that presents itself as a container, but it's a lot like compiler work.

The first step to transform program into code or into some form that permits optimizing it is to go away from the assignment form – the A=A+1 form – and into an SSA form, a Static Single Assignment – you distinguish the two values of A, the one before the modification and the one after. You assign two different names, which adds some temporal annotation to the container to turn it into the name of data, not what it contains.

So it is possible to go away from the container model while still preserving the illusion of containers, allowing users to deal with names like containers, only digging into the details, seeing the temporal evolution, if they need to, when they need to roll back in time. If we move away from the container model, and start to talk about the content itself, security becomes easier. We secure the actual content by signing it. For two decades we have known ways to do this that are absolutely unbreakable. Nobody can tamper with that content without making the signature invalid. The data is thus immutable – there are a collection of the versions of the content over time, and each one can be signed, and the integrity of each one of those can be assessed.

That's what CCN does. We get rid of containers. An CCN name names a collection of data. This is the content model. We're never naming a repository of data; we're naming sets of data. A set may have a single item in it, or multiple. Anybody can add to the collection. The set is defined by the naming structure. The collection – a set of items with the same prefix – presents a hierarchical naming structure. There is no single system of meaning. You can't describe an ontology or a hierarchy that captures all meanings; you cannot capture every way to represent a specific set of data, but the web provides an easy solution: links. A name whose value is another name, that connects one view of data, one way of naming data with another way of naming data. The connection can be anywhere in the tree. An ontology tree can be turned into an arbitrary graph.

11. What's in a Name

Describing naming for the calendar application helps make the concept more concrete. There's the prefix for the collection, a place to make names. For example, it could consist of the company, name of the user, and app – company/name/cal – then the thing that needs to be named, i.e., calendar items. The particular one that is being dealt with, e.g., a meeting, at a particular point could be some VCARD file for a card application. It's evolving over time if the meetings move around. If it changed three times, it would be labeled as "Version 3" of that meeting. That part of the name captures the temporal evolution.

You deal with things that are all different sizes in the world, but you don't want the network to deal with things that are all different sizes. The smaller the unit you can put on the network, whatever the maximum size unit, that's the granularity of sharing. And if you put really big units on there, sharing won't be very efficient. Between the objects and the network, you can use some fragmentation protocol – that's a model of TCP. Or if it's possible to name collections, then a collection of fragments is a collection. You don't need another name space to bust things up into pieces. You just name the pieces.

There's some parsimony, but also leads to very nice things happening in the protocol level. For example, s0 declares that it is Segment Zero and the first piece of whatever this item is, and that this item only has one piece. And the last is either the content or proxy for the

content. It's currently using SHA256 checksums. To name right down to the content level, specific bits and no other bits – this calendar item as it was today – those bits can be named. A link can be made to points to those bits. It can't be forged. It's always going to resolve to them, no matter how the entry is changed, because I put the bits in the name. It's a proxy for the bits. It's a finite-length proxy, but it's unforgeable because it's a one-way checksum. If I'm replying with the data, I can compute that proxy from the actual data, so I don't have to put it in the name when I'm returning the data, and it can still be validated. Names of this form are how you validate the binding between the name and the data. Essentially, you put the data in the name[xi].

The important thing is that this binding is immutable. I can make other objects like this, but they're new objects. If I change the calendar entry, I change the name because I have changed the bits on the end. I may or may not change the version number, but I create a new item if I change any part of it. Because of that, I don't have the coherency problem as I do when I talk about containers. Whenever I try to replicate content with a container model of the world, I have to make the replicas consistent with the master copy that lives in a container somewhere that lives on some machine that defines the content. CCN doesn't have locations for content; the content itself authenticates itself and identifies itself, no matter where it is. The user doesn't have to be consistent with any other copy; they only have to be able to validate the copy. To confirm it is a legitimate copy of Version 3 of the calendar item, the user just checks the signature.

This means that when you are doing coherence protocols, nothing can occur that can make what you know wrong. Nobody can change this version underneath me. You can create a new one to make my knowledge incomplete. But you can't make what I know wrong. And that completely changes the operation of the replication, the consistency and the coherence protocols.

There's some great work on system called PRACTI from UT-Austin, where they teased apart the replication of data from the storage of data, from the containers of data, and said if you could name the data, then you could give independent control of the replication and the coherence semantics, and you could do more than give independent control; each receiver could choose the coherence model that's appropriate for that receiver, which means a lot of the consequences of coherence, which are serial consistency, which forces you to do things really, really slow in a distributive system – you can make that go away. Your backup program doesn't need to be serially consistent, it just needs all the data. Because you've de-coupled the semantics of consistency, you're not dependent on a container, you're just dependent on getting all the data, you can use a system like PRACTI to make the consistency and coherence be receiver-specific. You can choose to operate in a lot of different performance regimes, and by default, you work in a very parallel very high-performance regime.

12. Signing & Establishing Provenance

The whole name includes the content. The thing that created that data, and the last thing it does, is sign the whole thing, and then it injects it into the network. That's done at the bottom of the app – the app's interface to the network – not at the kernel, and that's in order for the attack surface to be as small as possible. The creator of the data – the entity as close to the creator as possible – signs the data. It seals it, attests to its integrity. Nothing after that point modifies it.

With IP, packets get rewritten at every hop, so that the TTL can be updated. CCN packets are never modified; if they are modified, it is obvious. The program that created and signed the data has a signing key, and in the packet, there's a key locator, something that points off to the

key. The key can live in a packet, and it can be a pointer to the key somewhere else. It's generally the name of the key, and, in general, rule that says how to go from the name of the data to the name of the key.

Content-oriented models – where the data are actually named – do not have key distribution problems, which are the Achilles' heels of security systems. With a container view of information, and an armored-pipe view of the way information flows around, there's no way to associate the key for a set of data with the data itself, because there is no name for the data itself. Your'e not dealing with data being; an SSL connection gets the data. To get the key associated with that data, in CCN, you need the name. The name is the fundamental communication. And you can always algorithmically go from a name to the key for the name. You can say that the key associated with this name is the NAME/KEY. If you want to go get the key, you can present that name. Then, you use the CCN mechanisms to retrieve it, because that's just another named data item. So you get absolutely trivial key distribution at an arbitrary granularity. It could be at the data item level, the single-KB level, or the whole collection level.. You can choose, but there is no technical barrier to making it as fine as you want it.

My program signed it with a signing key that I gave it. I had a signing key that my employer gave me when they delegated that part of the namespace. In fact, the delegation itself gives me that key, which I use to sign the Calendar program's key, and the Calendar program uses its key to sign the data items. I'm not giving the Calendar program my signing key. I'm delegating it the ability. I'm giving it a delegated key – a longer key – and the reason for that is, you notice that the name of that key matches the leading name of the data item.

In our library dealing with collections at the application level, we use that convention explicitly. We say, 'that's the realm in which this key is valid.' So, this key can be used to sign a calendar entry; it can't be used to sign a check. It's a key, its meaning is a very limited domain. If it's compromised, well, I'm sad, but the damage is limited. That's coming for free. That's coming again because we've got these structured names. There are other things that can be checked from the same name.

It could go across Bluetooth; it can go across a company's network. It is does the latter, for example, the company network infrastructure can check if there's a data item that claims to be in the company namespace, it better be signed by a key that was authorized for that namespace. So, the network infrastructure is going to check that part of the signature, and say, 'Is the user of this namespace somebody that I want using this namespace, or is it somebody that has wandered in off the street and is trying to inject data into us?' If the company part of the key doesn't validate, the infrastructure can verify that the user is not authorized and will throw the data away.

That's the flipside of attack amplification. Now the attacker has to work against the entire network infrastructure. If somebody wants to inject bogus stuff into a company, the first company box that it hit rejects the data. Rather than having a lens, we have turned its around; the entire infrastructure defends us because the content is visible. What is being attempted is visible at both the data/item level and the certification level. The infrastructure can make the connections and use that to protect.

I'll use that as a segueway to the securiyt model.

With TCP, by design, the content is opaque. It can't help you protect it. All it can do is put an armored shell around it, and that can hurt more than help. We have this totally horrible

trust model, again, because of the communication – armored pipe containers talking to hosts – we try to do root-certified trust, which is outsourcing trust, which I consider like outsourcing breathing. Trust is how we live. It is what makes communities. But you don't accomplish it at the network level. You have to trust who Microsoft trusts or who Verisign trusts – whoever the root authorities are, signing certificates.

Now the original intent was for there to be only a few of those, quasi-governmental agencies working with a very high level of ethics, a very high level of checking. The current Microsoft Internet Explorer lists 263 root authorities from all over the world, mostly unrecognizable and difficult to trust. For the most part, however, browsers trust them implicitly. Any of those authorities can certify the certificate that claims to be a bank, and browsers believe it because root authorities assert it to be true – and not necessarily a root authority who gave the certificate to the bank. Any of the root authorities are equivalent, so any of the 263 guys that gave a nickel to Microsoft, they're an absolute source of trust. Anything they say is believed.

This is not a good world. There's way too much attack surface – way too many high-value targets. Data can't be named, and keys can't be named, so there is a completely unsolvable key distribution problem. Attempting to set up security policies – inside an organization, for example – people generally think in terms of what can be shared. They have a set of information, and they will share it with someone with whom they are collaborating, but not any other set. We work on information; it embodies interaction. But you can't express that kind of policy at the network level. The network doesn't know "what," only "who," so we have to express at the network level who can talk to whom. Your machine can talk to this machine via the firewall, but you have to be careful to put on that machine stuff you want to share. Manually trying to make the "what" congruent with the "who" doesn't work too well. I can't be a separate machine for every project on the firewall, and if I mix a lot together, the sharing might not be what I thinkit is. The information can diffuse because there's not a hard boundary, and I can't check it because it's not visible to the network. The network just knows about things at the machine level.

13. **CCN Signing**

In CCN, every packet is authenticated and publicly verifiable, and anything can verify any packet – which is very different from current practice. The best current practice in a TCP/IP network – although rare – is what's called HMAC, a Hashed Message Authenticity Checker. For example, if there is a secret, such as a shared random number that a set of participants know, you concatenate with you data, then you can compute a one-way checksum of all the data plus the secret. This is a one-way checksum, you cannot go from the checksum back to the data, so only people who know the secret can authenticate the data. The problem is that it's completely symmetric. Anybody who can verify can also forge. There's only one secret, and only the people that can verify can attest to whether or not there's data integrity. No other party can – particularly, the network can't.

CCN does something different – it actually signs the packets. You compute a one-way checksum of the packet plus its name, and then you encrypt that with your private key, and in the packet you put a locator for your public key. Anybody who gets that packet can take the public key, decrypt the checksum, and then separately compute the checksum from the data. If those match, they know that the data hasn't been modified. And the fact that the public key, your

public key, decrypted the checksum proves that you signed it. Not only does this provide message integrity, but also authenticity – from the public key crypto.

Unfortunately, computing a signature is expensive, or some measure of expensive – somewhere between 100 microseconds and 3 milliseconds, depending on the algorithm that is being using on a current generation Pentium, a 2 GHZ Pentium. But you don't have to compute a signature for every packet. There are options, such as the Merkle tree[xii], where you can have one signature for a whole collection of data, as well as incremental computations that go under that signature physically in the packet. The packet has to contain the log number of those. If there are 1000 items, there would need to be ten partial computations in every packet, but only one signature computation. It's very cheap, so there is a very high level of amortization.

Similarly, ticket-signing mechanisms can make the signing operation arbitrarily cheap and verification relatively cheap, as well as probabilistic. Receivers choose what they verify and when, e.g., they could verify the first packet and trust following ones. They could choose to verify every nth or some random number; they choose their amortization schedule.

With public verifiability, if you allow content to be replicated, if it can live at multiple places. A problem arises called the "Chinese Telephone Problem," where each copy can pick up errors. And these errors are persistent. They live in the replicas, so with each replica, there are more errors – all the original errors plus new ones that came from the transcription. You need to be able to get rid of replicas that contain errors, so they don't live in the network. Otherwise, they'll impede communication; they waste space. Attackers can inject data where they pretend that it's signed, but it's not signed. They don't know the keys that were necessary to sign it, but they can put metadata that looks like it's got a correct signature, and without a check, it's not possible to detect a forgery. It is best not to do that at every node in the network. It's better to have a system where end nodes – the consumers of the data – check the data, and if they find that it doesn't check, they can go back to the infrastructure and declare the data bogus. The infrastructure doesn't know if you're telling the truth or lying, but if it is publicly verifiable, they can check, and can then declare the consumer trustworthy, able to declare something bogus in the future without having to be checked again.

Reputation systems and gossip systems can distribute the work of verification in the presence of replication, so that valid replicas everywhere do the work at the edge, providing scalability. As the edge grows, you get more places to do the checking, with less work concentrated in the center.

14. Trust Model & Key Distribution

So you have to do security validation in some sort of trust model. In the current model, the gods anointed by the root authorities determine all trust, and that's a really bad model. Rivest's very nice certification system, SDSI[xiii], for example, led to some RFCs being written, but it didn't have the impact that it should have, in large part because it's a model that's focused towards content, but IETF Networking isn't about content[xiv]. SDSI is, however, simple, extremely robust, and completely distributed. The choices are completely local – you don't have to coordinate with anybody or anything. It's been fully axiomatized[xv]. You can express it in formal logic, and you can validate that logic against standard security models and standard threat models. You really can depend on it at a first principle's level. Rivest's model and Joel Halpern's axiomatization are what CCN uses.

CCN uses a lot of keys – signing keys for authenticity and encryption keys for privacy. You can make the keys largely automatic, a benefit of the name; it's easy to tie keys to data. With a new set of data; the policy says it wants to be private. As part of making the set of data, I generate a random number, and that's the key. Then, I encrypt that key under my public key, or under group keys, and spread it throughout the world so that they're replicas and they can't get lost – and now that data is private – and it is made private. It never lives in the clear. In normal networking, that would give you an unsolvable key distribution problem, but CCN is perfect for key distribution, because you can name keys, and anything you name you can distribute, and they're replicated so you get very high reliability.

The SDSI trust model has these keys – particularly signing keys – and the reason one person would trust a key is not necessarily why somebody else would trust a key. A reason to trust a key acquired from a bank, for example, is that one's business relationship with the bank led to exchanging keys. You get a credit card offer from my bank, and you look at it and say, 'Well, did this really come from this bank, or is it from some scammer?' You can check the key in a registry of keys; you can check with your friends to find out if they've seen that key before. The way that you establish trust in that key has to do with your context for the information – how you value the information. As opposed to today, where there's a one-size-fits-all: We check with Verisign and see if they signed the key.

15. **Identity creates organism[xvi].**

There's a fundamental change that happens at the network management level and the network configuration level when you put security at the bottom. The thing that is fundamental to making an organic unit, a cooperative unit, is a shared notion of identity. Human identity comes from DNA; it is in things like bodies. But it also comes from other things like communities and families because of shared context and shared experience. You can't make networks today that behave organically, that have levels of cooperation in the network, because there is no notion of identity that could be used to bootstrap that organism, that cooperation. If you move to a content view, several things get easier.

One, configuration is dead-simple. The signing key is the only thing that you ever need to configure. It includes identity at many levels; it's the structured key. Many pieces of communication embedded in that identity can be used to establish who you should trust and at what level. But also, with host-to-host communication and point-to-point communication, there are always two ends, and they have to be configured to agree. If the same thing has to be configured in two places for stuff to work, it's likely not going to work, as the chances of misconfiguring increase. Without point-to-point communication, while talking *about* stuff, not *to* stuff, all configuration is single-point. You don't need agreement. The other side has to have it in order to respond, but nobody has to agree. So there's a really robust configuration – single-point configuration – that you can't misconfigure. The configuration includes some notion of community and identity, and you can use that as a sense of self.

If you're learning about the world, about the things around you, and about what you should do in that world, when the outside world tells you things, gives you information, tells you about its context, you know what you can believe and what you can't believe. You know what is relevant to you, because you can filter it through your identity. And they have to put their identity is in everything that they're telling you. So you can say, 'I'm a router and you're giving

me adjacencies, and we're the same organization, we're the same network, I guess I'll believe your adjacencies. If you're not the same organization, you're not the same network, you've got a different signing key, well, I'm glad to know about your adjacencies, but they have nothing to do with me.' So you get an easy way of getting trust without additional configuration, and the example that I gave about defending a namespace – all of the infrastructure can recognize 'not self' in addition to recognizing 'self'-- the two concepts are complementary.

You can muster all the infrastructure to protect things that need protecting, rather than having to localize those. We can't do that with current networks, because security is an afterthought. It should be fundamental, as it's fundamental to organisms. Security is what gives identity. The tendency is to think of it in terms of privacy, in terms of encryption. But the work of work should be on signing, because identity[xvii] is a much more fundamental and more important operation – a Swiss Army Knife of security. Privacy is interesting – being able to encrypt stuff and keep it private – but it's not nearly as fundamental. And identity doesn't have to disturb privacy. In fact, it's the other way around. Signing can create anonymity[xviii]. You can sign money in a way that is not traceable; it's identified as money, can only be used once, but you can't tell who used it or for what. That's again, a first-principles assertment – a very hard guarantee of privacy. It's a mathematical guarantee, and nobody, not even Dick Cheney, could tell who used the money to buy what. No information would permit reversing the spending operations and tracking it back to who spent it. This is something that should be integrated into our thoughts very early, but it's not. At least the tendency in the US is for people to say, "I don't do security. I don't know security. All that security stuff is magic." It's a poor point of view that really impoverishes the world.

16. **Node Model, Packets, and CCN Names**

Again, CCN has a different view of content – of the content itself, not its containers. This can be tied to the moving parts – how the nodes work. It's important to know what the packets are that it deals with. There are two kinds. There is an Interest packet, which contains the name and a little bit of other goop. There's the data packet, which is what you respond to an Interest with.

An Interest is asking a question. It's sort of like an HTTP/GET. "Does anybody have any data in this collection that matches this name?" The data is an answer. And the data, because the binding has to be secured, include the full name, the actual data, the signature information, and then some other goop. This whole thing is coded in binary in something that can be one-for-one transcoded to XML so that multiple units of data can be lumped together. And any of those fields can be extended if desired. In particular, an arbitrary amount of security annotation can be added.

CCN names may seem like comfortable, human, readable names, but they are actually not. A name in CCN is an opaque object, and the only thing that CCN cares about is that it's a hierarchically-structured object. It uses the structure, but there is nothing else that it knows about. This (slide) is how a name is represented; it's a count of the number of components. It could be "one" if self-certifying flat names are used. It's probably not a great idea, but it can be done. One component is fine. For each component, there's a count of the number of bytes in the component. So, it's arbitrary binary data; the only assumption is that there is hierarchy. Encrypting a name so that people can't see them is fine, as is arbitrary binary data like the packet

signature, the SHA256 – that's a real SHA256, not an ASCII-coded version. It's already taking a lot of bytes, we don't want it to take anymore.

Why hierarchy? Same reasons as IP: longest-match lookups let you aggregate state. It goes far away from places where the content might live. You don't have to know the details, only where to findmore details. Longest-match lookup let's you find the longest prefix that matches the address that you've got. With IP, that gives log(n) state scaling in the core. As the network grows, the state grows only as a logarithm. Even though there are 3 billion hosts on the Internet, there are approximately 300K routes in the core. And that's a completely tractable set of state, even for a global network. Of course this only applies to globally-routable names. Similar to how some IP addresses are not globally routable – net10 addresses, or 192.168 – they don't even appear on the routing tables, CCN has an equivalent: local names.

CCN uses longer names, often derived from human, readable names, and they might have content identifiers on the end of them. And they're variable length. At first look, it may seem as though it's going to cost a lot more to do packet forwarding because packet lookups are going to be more expensive. But as it turns out progress in (theoretical computer science) hashing over the last decade has just been absolutely astounding. It has been demonstrated[xix] that it is theoretically possible to compute collision-free hashes in linear time on a billion elements with the cost of about 6 microseconds per element and 99% load factors in the hash table. That's a hash table with a billion elements – collision-free, with a single-probe always getting to the right element, and all the space is used, no inefficiency. Years ago, it was assumed that was impossible[xx]. A hash table could not be loaded more than 30%, and there would be collisions. Light years of progress have been made in hashing[xxi], but none of it can be used by IP, because the structure of an IP address isn't explicit in the address.

The way you find out what is the net part and what is the host part of an IP address is to look it up – that information is kept in the routing table. That means you can't hash the address to look it up, unless you hash it a bit at a time, which means for an IPv4 address you do 32 lookups, one for each prefix of the address. That's possible, but a lot of lookups. Without knowing the prefix, it is not possible to hash the address itself. The only way to find out the prefix is by looking it up.

Which means that IP lookup structures – particularly for IPv6, where there is no option of doing it a bit at a time – often tend to be radius tree structures, where you have to do a treeWalk, and when you hit a terminal node of that tree, you've found the longest match. The routing table reveals the address structure by terminating the walk. Or you can do it fairly fast in software, at log(n), cost, but it's still a tree walk, a lot of memory accesses. Another option is doing it in hardware, using a Ternary CAM, then you can do the match in parallel, but at a fabulous transistor budget.

So, DRAM is one transistor per cell, yielding big memories, Gigabyte memories. SRAM is 12 to 14 transistors per cell, which is really fast, so you can make the memories that sit right on the end of the serial to parallel converter for very high-speed lengths, but you pay an order of magnitude-density cost. For the same price, you get a much smaller SRAM. TCAMs are 100 transistors per cell, because you have to put the matching logic and priority logic in every cell. It is two orders of magnitude more expensive or two orders of magnitude less in density. Very big routing tables in TCAM aren't possible without spending a lot for the routers that use it. This is a property of the way IP lookups work – the structure is implicit and not explicit. We wanted to fix

that for CCN, we made the structure explicit, so we can use hashing and the last decade's progress in lookups. Even though the names are longer, you can look them up as fast as IP names.

What does all this mean? Well, one of the big successes in IP is that the names don't contain their meaning, so the router has a set of rules for dealing with names. It does a longest-match lookup to find them in a FIB, a Forwarding Information Base**.** But the meaning of the name, where it goes, is completely determined by the contents of the FIB. That has let us evolve a lot of different meanings, including "this host" address, the 127.1 address. That didn't require any change to the IP forwarding model, but it caused us to set up a FIB in a certain way – 127.1 is never sent to anybody else. It isn't accepted locally, and if it is heard from the outside world, it's dropped on the floor. Simiarly with local-use addresses like Net10, 192.168. Such addresses are deliberately not routable. If there's an attempt to push them across the boundary, the other side of that boundary refuses to accept them. They have different meaning in different networks, so they can have local meaning.

Some addresses have global meaning – NET18, NET13, they mean the same thing all over the world, defined by the way that you set up the routing table. Therefore, the semantics of addresses come from policy considerations in how you set up FIBs, how you do forwarding.

It's the same for CCN. The node machinery doesn't give any meaning to addresses. In particular, addresses can have local meaning that you can re-use. For example, in order to avoid a cable dance, a presentation would have a globally-routable name and you could pick it up anywhere in the world. In the room of the presentation, there'd be a projector, which could be spoken to by the presenter on a local name that's the same in every room. If every room has a projector, the presenter can talk to each one, i.e., "this room/projector." It could be wired into a piece of software that I never change. To give a presentation, the presenter could say, "Projector, here is a presentation. Put it up please." There would be no wires to hook up; it'd be a name that re-hooks itself up, a contextual name. And we can imagine variations on that theme: the room, the meeting, a family – names that have useful local meaning changeable based on the context. The name can be made static, and it can take advantage of the fact that it's being re-bound.

In particularly, coupled worlds are easier with CCN. For example, how would one know the IP address of a light switch? There's no way to express it. In a content model, it's easy to give names, and if there's a communication infrastructure, it's easy to talk to names. Particularly relevant to the current revolution in lighting – going from slow florescent and incandescent lights to LEDs, which can be modulated quickly, Gigabytes per second. You actually want to modulate them, because of how the human visual system works – the phosphorus in eyes is fairly slow. If a light is turning on and off faster than the "flicker fusion frequency," it is perceived at its brightest intensity. It being turned on and off is not seen, and the average intensity is not seen. Rather than just modulating it with a Sine wave or some random frequency, why not put data on it? The room can announce what's in it. Cameras on laptops and ambient light sensors in phones, etc. – can be reading the lights and learning about what's going on around them.

We're deploying all sorts of new, local data-capable infrastructure – ideal for distributing content and providing more awareness of environments, as well as closer coupling with the environments. These devices are not very amenable to an IP model, but they are to a content model.

17. **Operation of CCN**

The basic mechanism is dissemination, a broadcast model. To get data, you broadcast – logically broadcast – over a point-to-point link, an IP tunnel, anything you want. You state the Interest and give the prefix of the data you want. A collection of data is the stuff that matches that prefix. Anything that can hear the Interest item can respond, if it has the data item. That's not the way Google works, but is the way that IP works.

With IP, When you put a node on a network, it does a broadcast, an ARP that says, "Anybody interested in sending packets to this destination, send them to this MAC address." It binds the logical identifier, the IP destination, to a physical identifier, some interface MAC. Routing can propagate that, not the host part of the address but a prefix. They need to know that binding to be talked to on the local Net, but they don't have to tell it to the rest of the world about it. Essentially, you are getting Interest, in the CCN nature of Interest we call them routes, but they flow out through the world, and packets follow those – a trail of breadcrumbs that cause packets to be delivered.

A host – 1234 – it ARPs, it broadcasts, it states that it is that IP address. It's got a couple of routers on the boundary. They look at the ARP, remember where that host is, and remember this MAC address, so they can talk to it. They're in charge of Subnet 1.2.3, so they don't need to advertise the 1.2.3.4, because the 1.2.3 identifies anything whose address starts with 123. The 4 is redundant; it's irrelevant. They are just propagating their prefix. This network is stating its network identity, 1.2, so it doesn't have to propagate Subnet-level detail. The 1.2 gets distributed via the whole Internet. When anybody wants to send to that host, they send to 1.2. It gets to the border router; it can look at the destination address and says it needs to go to Subnet 1.2.3, so it sends to that Subnet, and then the border routers know what the host is. Fine-grain information close to the host; coarse-grain information far away. That's what gave IP nice scaling. That's how CCN works. No different.

Transport-level function: An Interest is sent out. There's some matching Data coming back that makes the Interest go away. As the Interest is flowing out; it's laying down breadcrumbs. That's just like IP routes, but CCN has the model is as data is coming back, it consumes the route. It eats the breadcrumb, so that other Data items can't follow it. A new Interest would need to be expressed in order to get additional data items. If there is lot of data in progress, a big bandwith/delay product that you need to fill. you can express several Interests simultaneously for different data. All of that data will be received simultaneously. That's similar to a TCP window.

In fact, this whole mechanism is really similar to a TCP window. With a TCP conversation, the sender sends until it fills the window. The window tells it how much it can inject into the network, and then it stops. It has to wait for an ACK before it can send new data. Generally, you get an ACK in for a packet, and that allows the sender to send a new packet; it exposes another chunk of space. A sender that is turning ACKs into data packets, which basically consumes the ACK. And a symmetric thing is happening on the receiver; it turns data packets into ACKs. That essentially consumes the data packet. They're working in a pairing, a flow-balance. There's a beautiful piece of work by Frank Kelly of Cambridge – a book he published in '79 – which I didn't read it until '89. Would have saved me a hell of a lot of time if I had gotten to it a decade earlier. It's called 'Reversibility and Stochastic Networks.' It talks about the properties of this detail balance.A network with this pairing, with this balance[xxii], or

even approximate flow-balance or some fine transform of a flow-balance, then the first principle's properties are robust. It's proven stable under arbitrary traffic and arbitrary load. It's basically why the Internet can grow. It doesn't matter how big it becomes; it doesn't matter how many hosts are put on it. The balance property lets all of the interested parties control the traffic, so that it doesn't get too far out of balance. There's a bound on how bad it can get. That bound can still be pretty bad, but it's the amount of buffering that is in the network, and there tends to be a lot of buffering put in the network so things can suck pretty badly while waiting for all those packets to go through, but it's a bound and a hard bound, a hard mathematical bound that says, "You can grow this network."

This flow balance a really important property, another one of those things that should be taught but isn't. If you're designing protocols, the flow-balance and the symmetry properties it implies are of particular importance. TCP has used it to great effect. CCN is using it for the same reasons. TCP uses it in a unicast scenario, because it's point-to-point conversation. But the nice thing about Frank's work is that it works at the traffic level – it's "can you pair a packet with another packet?" CCN pairs Interest and data, and both of them are broadcast. And even with multicast scenarios, it's still stable. For decades, we tried to make multicast versions of TCP, and we failed – it's really hard to make multi-cast transport protocols – and part of it is not understanding the underlying dynamics, and the balances you have to maintain. We can show that balance for CCN formally. We haven't shown it in simulation but we have in a prototype implementation. What Frank said works.

18. **IP Machinery**

It is straight out of RFC791. There are interfaces that are point-to-point wires. There's a network core with a table in it, called the FIB or Forwarding Information Base, that says, for a given destination address, where you should send the packet. Because of stat. muxing**,** the interfaces need a buffer in front of them. Because there is no global scheduling system that prevents packet conflicts, it's perfectly reasonable to have a packet arrive on interface 0 and 1 simultaneously bound for interface 2. Even if all the wires are the same speed, you will instantaneously overload the output interface by a factor of 2, and there's no requirement that those interfaces be the same speed. Very often, if some are local interfaces and some are long-haul, the locals tend to be many orders of magnitude fatter and higher-bandwidth than the long-haul interfaces, because of the economics of bandwidth. The design of the protocol requires a buffer to handle the transient overloads and the rate adaptation.

The packet-forwarding model is: The packet comes off the wire. The first thing to do is to ask, "Is this destination address me?" If so, you punt it up to the transport stack. If not, you need to prevent loops in IP, because it's not aware of the data. Every packet has a time to live, which is basically a loop-killer. You decrement it at every hop. So you check the time to live. If it's non-zero, you decrement it and then forward it. To forward it you do a longest match lookup of the destination address. The FIB contains an outgoing interface, and you put it on the queue for that interface, and that's the packet processing. That FIB can be exactly one outgoing interface. In IP, you have to forward on a tree – the gravity property, because loops would kill the network. The normal way to architect that is to overlay a spanning tree on it, no matter how rich the network graph is, and forward via that spanning tree. The tree has the property of only one path between any two nodes., so there is only ever one outgoing interface in the FIB.

19. **CCN Machinery**

Here's the CCN node model. The interface is the broadcast. They're not wires. Abstractly, they're radios. An Interest could be broadcast on them. They can be point-to-point links; they can be tunnels over IP at any point in the world. They can be applications. In CCN, there's no difference between demuxing to a local app and demuxing to some foreign agent. It's just demuxing. There isn't an "is it me?" decision. It's just demuxing to a local app. You need that with multi-point protocols – because there may be a service agent locally, but the data may be available multiple places. You want to ask for the local copy, but you want to ask others. Whenever you have a protocol where you can fan data out, or in, then if you do the classical 'layer 3/layer 4' split, you end up hosing yourself, because you have to either send the data up or out. And you can't send it both ways. This is not how the protocol stack is organized. With many-to-many broadcast communication, you need to put applications in the same framework as everything else. That simplifies stuff.

There are three different data structures. One is a content store; which stores data seen before. It is the same as buffer memory in a normal router with a different replacement property. It uses LRU rather than MRU replacement. It is not treated as a FIFO, but it remembers the packets. It can verify all of the packets, to get rid of the corrupt ones. There's an index structure – let's yo go from the name into that content store. When you send Interes, it lays down breadcrumbs, and the Data follows those breadcrumbs back, and it consumes them in the process. That's embodied in the pending Interest table. It's an Interest that couldn't be satisfied and has been sent on to somebody else. But you remember it, so if data are received, you can give that Data to those who originally made the request. It's got the name from the Interest and the face that the Interest arrived on, so when data arrives, that's the face that the Data's going to go out. That's the breadcrumb. And for every Interest that you can't satisfy locally, but you forwarde, there's going to be an entry there. Then, lastly there's a FIB, which looks just like an IP FIB, but rather than having a single outgoing interface, it's got a list – because in CCN data can live multiple places. What you put in the FIB are all the places where the data might be.

In IP, it's a different semantics – the output interface is where the data must be. When a prefix is announced, the contract that the routing protocol is making with the network is "everything covered by that prefix is reachable via me." That has to be the contract, because the people that you're telling the prefix can only send the data one place. If they send a datagram with an address in it, it better be possible to handle that datagram. You have to get it closer to the destination, because they only had one choice, and you were the choice. You get global dependencies in IP routing because you have to put on a spanning tree – where you have to be able to handle all the traffic covered by a prefix. There's no such global dependency in Content Routing. If there are three places that the content might be, then CCN is perfectly capable of asking all three. It can ask all three simultaneously, and two may not have it. Only one needs to answer, so there aren't global dependencies, only local dependencies – a much weaker contract.

There are these three tables. You still only do one lookup. An Interest arrives. There is a single-index structure. For example, imagine something on a wireless network trying to pull down a video, with the following name: parc.com/video/WidgetA.mpg/v3/s2. It's version 3 of the video, and you can see here that we got block zero, the first block of the video; it's in the content store, and block 1 has been requested; It's in the PIT, but no data has come back yet, so

it's still pending. And the thing that's asking for the video is asking for the next block, for block 2, the third block of the video.

That Interest arrives. We look up the name in the index structure – our longest-match lookup. It doesn't match anything in the content store, because there's no S2, and it doesn't match anything in the PIT. There's no S2; there's an S1. On longest-match-lookup, it does match the parc.com in the FIB. So, we don't have the data that the application is interested, but the FIB says they might know where to get it. And so we forward the Interest by taking the face list out of the FIB. The question arrived on face zero, and you never send a question back out to the face that it came in on. That's flooding semantics. It's just going to loop. If you're going to forward it, you remove the face it arrived on (0), and send it out Face 1, presumably to some server at PARC. That's going to bubble it upstream. You create an entry in the PIT that says, "I'm asking for parc.com/video/WidgetA.mpg/v3/s2." So it says, "I've got a second pending Interest – one that's being propagated." A packet came in – a single lookup – it can be three different places, but they have a priority order. If you've already got matching content, you always want to return it. If you get a Content Store match, you satisfy the Interest. You send that data out the wire and you're done, throw away the Interest. If you get a PIT match, then this may be a new Interest in the same data – somebody on a different face is asking for data that you're already trying to get. If that's happening, then you simply add that new face to the PIT entry, which says, "Okay, I'm getting it, and when it comes in, you'll get a copy as well as the guy that asked the first time." The requesting faces, that's another list. If you get a new PIT entry for data you've already requested, you add it to that list, and now you throw away the request because you've remembered what you need to remember. If it's a duplicate, or if it's coming in on the same face, it's a guy that's impatient, and he's asking you again – or somebody that didn't hear the first request is asking again. And when the data comes in, you're going to satisfy their Interests, so you throw away the Interest. You don't need to do anything. There's a priority. You look it up in the content store. If it's not there, look it up in the PIT. If it's not there, look it up in the FIB – one lookup where you've ordered the entries in the lookup, so that the entries arrive in priority order. That's easy to do with a particular kind of hash table, a Treep, a priority tree. There are probably other structures that do it as fast, but it is hash; it's an O(1) lookup – one lookup, same cost as IP, more data structures.

When the Data packet arrives, so we send up the Interest at that interface, the data packet comes back in. We look up data packets only in the PIT. Well, that's not strictly true. We look up the data packet. If it's in the Content Store, it's a duplicate, and you throw it away. The data is idempotent – looking up the data packet, means up to and including the data itself. So, if you find it in the Content Store, you've got it, unambiguously – up to the SHA256 checksum – you've really got this data packet, and so you don't need the duplicate copy because you know it's a duplicate. You toss it. What that means is that content can't loop in CCN. If data goes around a loop for any reason, as soon as it gets back to a place it's been, it's thrown away. So duplicates are suppressed at the content level because it's a content protocol. So you can use a diffusion flooding paradigm, because the data is recorded. We look up the packet; we find it in the content store; it's a duplicate; we discard it . If we find it in the PIT, it's something that we asked for.

What was this one? This was S1, so we do find that in the PIT. So, somebody asked for this. What that means is I record the packet in my content store; I got something that was asked

for; it's relevant data. And for each interface listed in the PIT, I send the packet out that interface. I add something to the work list of the outgoing interface that says, "Ship this packet that's now in the content store."

I say it that way because in CCN, there aren't queues. Conceptually, the model is a router, an upstream node, always satisfies requests from its content store. What you're asking for is something that's in the upstream content store. If it's not in its content store, but there's a way to get it, then recursively, it will ask its upstream for that stuff, put it in the content store and then send it to you. But, you're always satisfying requests out of a memory. The memory in CCN is explicit, rather than being an implicit buffer memory in IP. Because of that, you don't need queues on any of the interfaces, because you don't have to do rate matching. They get the data when they want to get the data – out of a Random Access Memory – and they get it in the order that they want to get it. So the service policy is not the sequential policy that is motivated by queues, but it's an instantaneous policy per face that says, 'okay, what data do I need to send *now* based on the downstream demands on me, and the downstream contract – so you don't have the ordering implications.

Note, roughly 50% of the QoS problems on the Internet are artifacts of queues: the fact that a queue introduces this big delay and it's serial. you can put stuff on the end of a queue, you can put it on the middle; you can't service out of the middle. If you get rid of queues, then you get rid of that 50%, because the service policy can use the fact that you're servicing out of a RAM. It can be much more responsive to the downstream demands, and to the instantaneous changes in those demands.

20. **Comparisons**

These are the same parts that are in an IP router; they're just combined in slightly different ways. The buffer memory turns into the content store. Same memory – same amount of memory – you're just changing the replacement policy. Goodness is not in a queue in FIFO buffer. You want to keep it empty, because it's a shock absorber, and if it's not empty, it can't do its job. You forget what's in the queue as quickly as you can. That's good for the TCP protocols, because once you've sent that data, it's of no further value to you. But given the relative cost of bandwidth and memory, actually the stuff in that memory is really valuable, and you don't want to spend bandwidth to refill it – so the model in CCN is, 'no, if you've got it, remember it. And use it again, so you don't have to spend more bandwidth to get another copy.' Once data arrives, you hang onto it as long as possible.

That's a different replacement property on the memory. It's still a memory, but with an LRU replacement, as opposed to trying to forget as soon as possible. But the packet buffers. You post the buffer to a face; the face fills it with the bits that arrive, and then, you look up its name in this index structure, and if you want to remember that's the data packet that's arrived that's satisfied some Interest, then the way that you put it in the content memory is by putting a pointer to that packet – the one where the bits landed – You put a pointer to that memory in the index structure. You don't copy anything – this content memory isn't like some database. It's where your packets live. You're not moving stuff around, you're just remembering them in this index. The really new thing in CCN that's not in an IP router is an index structure, so you can reuse stuff.

Because it's multi-point, a lot of the transport state turns into the PIT. It's because the data could need to be replicated[xxiii]. You need to be aware of the local connectivity in order to duplicate data. A lot of the layer 4 stuff doesn't make sense. It has to move down. There isn't a "me" test for the same reason. If multi-point, "me" just has to be completely symmetric, the same as any other interface. So you forward to apps the way you forward to anything else. You never modify data – there's no TTL check – nothing can loop, so there'd be no reason to do it, and we don't want to modify packets because it means that we can't assess their integrity. That means that if you're doing particularly a silicon pipeline, you can do it a lot faster, because, a lot of the real estate of an IP forwarding ASIC is due to the TTL update. You have to rewrite the packet, so that really changes the flow-through semantics. It's not just the TTL, because when you change that, you also have to change the checksum. So, there's some computation, and several fields you have to update. None of that happens in CCN.

21. **Routing and Transport. Faces vs. Interfaces**

In CCN, there are no protocols like ARP or IS-IS. IP requires Layer 2 point-to-point identities, because that's what it speaks.; that's what an IP address means. So it does binding operations to set those up; you don't get those in CCN. Those are two simplifications. There's a complification. You have to handle multi-access interfaces, i.e., broadcast interfaces, and the canonical problem on a multi-access interface is a success disaster. You ask for some piece of data, and a hundred guys that have that piece will try to provide it at the same time. It's called an ACK implosion. The MAC-level protocol will resolve the contention so that they don't put their bits on at exactly the same time but it's a huge traffic burst, and it's doing nasty things to the MAC. It's driving it into a deep congestion state. Long ago – when we were doing multicast protocols, it became clear how to avoid that – suppression mechanisms.

So you're broadcasting a question; you want to do the transport equivalent of CSMA, which is to randomize the responses over time and have the responders broadcast, so all of the potential responders can hear the first responder, and they then turn off and state that they don't need to respond because somebody else already did. Rather than having everybody respond instantaneously, you want to spread them out. If there's a lot, you could could spread them out over a long period of time – because of the birthday problem. To make sure that only one responds, you must make the time interval the square of the number of potential responders. If there are ten guys, you must wait a hundred time units in order to not have them conflict. That's grim.

To get around the birthday problem, you partition the responders; you make a rule for who should respond first, so that within the partition, the numbers are small, and so the n-square doesn't bite you. In a partition, if there's one guy, it's a big win, because 1-squared is 1.

So there's a simple prioritization rule for CCN. If you created the content, you're very high priority to respond. You're the guy who made it. You should get to answer questions about it. If you didn't create it, but you learned it off some other wire, the hypothesis is it wasn't created on this net, and if it was, the guy who created it will respond. If you learned it from some other wire, then you're second priority to respond. If you got the data via some other source, or you learned it in another environment and brought it here, you can source it out of the Net. But if you learned it on this wire, you're lowest priority to respond, because many other people learned it on the same wire, so you should randomize over a big interval. With that priority rule, you can

do a simple suppression, where if the content is being created on the Net or being sourced on the Net, there's no delay at all. The creator just immediately sends the data. If it's being gatewayed from some other net, the gateway responds after a very small delay – with the content, just making sure the creator is not there. If it was learned on this net, but the original creator or the gateway went away, then anyone who has a copy will give it to you, but after a substantially longer delay. But you only do that for the first time.

There this name structure. It can remember stuff about this name prefix. So you remember in the prefix, if you responded last time, if you were the one that won the suppression game last time, then you get to send immediately next time. And you get to do that for N many trials. So you amortize the cost of doing the suppression by essentially doing an election once via a contention mechanism. After that, you use that election until you run out of data or it's time to run a new election. At that point, you run a new election. Therefore, you can deal with this broadcast interface as if it were a point-to-point interface. You don't have to deal with contention issues, because you've got some memory.

22. **QoS**

The quality of service problems in the current Internet – they are not distributed throughout the Internet. A bad Skype call, for example, is not losing a little bit at a thousand points along the Internet. It's a bad call because it is losing at one point in general. There's some exchange point, peering point, choke point, return path, tail coming out to caller, tail going to the other participant. There's some fast to slow transition, where a lot of traffic is piling up and stuff is getting lost. The only places there are queues are where there's a fast to slow transition, and there aren't many of those. Knowing that doesn't do you any good. It doesn't help fix the problem. The call still sucks. And it's because the communication model in IP is sort of an action at a distance model.

We want the abstraction that you have a single wire between the source and the destination, and congestion is a property of that virtual wire. And its components, its individual elements, are not visible to the end points. There are tools – pathchar and traceroute – that will shed light on what's going, but they don't help fix it. For IP, the observation that congestion tends to be localized properties of the engineering and the infrastructure doesn't help. That localized congestion is generally half the problem. There's another half the problem that comes from IP's unidirectional model. As an example of this: a kid's addiction to computer games.

The kid never sleeps. He grabs food from the kitchen, goes up to his room, and is just continuously immersed in these games. And the games send huge volumes of data over the family's Internet link. And if it's in the US, it's a slow link. The parents are trying to get work done over that link, and they have to compete with his goddamn gaming traffic. They can't tell their ISP that the work traffic is more important than the kid's gaming traffic – because the queues are in the direction from the ISP to them. But the Interest, the value of the data in those queues, is in the "them" to ISP direction. But there isn't a channel that goes from them to the ISP. There are channels that go from them to the ultimate destination and back again. There isn't a local communication channel where they can say, "We pay the bill. Send those bits first," or "Send his bits last." There isn't a way to express that.

Again, in the CCN model, all communication is local. You're talking to an upstream, and it's a bidirectional communication. You send an Interest; it sends a Data. Some of the annotation

that goes in the Interest says, “This data is more important to me than anything else I’ve asked for.” When you get data that match this Interest, and you have a choice of ten things to send, send this first. Because the upstream agent is in the protocol, it’s the thing you're sending the Interest to, and you also communicate your utilities to it. And because there’s this flow-balanced Interest-Data communication, it can react to that loop. It can say, “I’m pulling this stuff out of the Content Store, and I don’t care about the order, but my customer told me, he cares about the order, so that’s the way that I’m going to pull it out.”

The local communication provides a channel to remediate this problem, the lack of a detailed control over the bottleneck link, because both ends are explicitly modeled. It’s not action at a distance; it’s local action. And each of the links is treated differently. People have a business agreement with their ISP. And providers have a different agreement with their ISP. And over that link, they have their own utilities. They can be providing telephony service, which they rate-limit on admission control. That’s their agreement. They don’t have to have the same agreement, because it’s a different conversation. And because it’s local, and because it’s pair-wise, it can reflect those local agreements. It’s not a one size fits all, and it’s not an action at a distance agreement. And the part of the problems that have to do with queues, there are no queues, so you can just punt those.

Whenever you're aggregating, aggregates can create problems, so it’s important to service the aggregates. You do that the context of a business agreement – either a customer/provider business agreement or provider-to-provider business agreement. And if you model the individual links, that’s easy to do.

23. **Routing**

Routing has recently become an area of active research. There are very exciting things happening – very scalable, locally-controlled, local model routing. The most popular ones are various ways of doing small worlds – either explicit small worlds, hidden metric space small worlds routing, or more explicit small words, Kleinberg's hyperbolic routing, where approximate, incomplete local information, can get very good routes with very low stretch factors, so it potentially has great scaling. You can have lots of information, but mostly you don't need to know it need to be known. You just need to know the stuff that is around you, and it doesn’t require summarization. There are more viral models, things that handle very high degrees of mobility, where you are doing stuff that is based on either gradient models, stuff that you should avoid, or percolation models, random walks with learning so you can find hubs and good distribution centers.

It’s all very cool, and it all applies to CCN, as well as to IP. It’s semantically pretty equivalent. It’s generally easier to do in CCN than IP, for a few reasons. You don’t require conversions in CCN, because it’s not possible for things to loop. So you don’t need a stable topology; you don’t need short times to a stable topology, because those exist to prevent looping. And you don’t care about looping. The fact that you’ve got multi destinations means that you don’t need perfect state. IP requires a certain minimum amount of state, because you have to distribute packets along its spanning tree. You can’t send packets along the spanning tree and somewhere else, because that might cause a loop, and loops really rapidly take down your network. And you can’t send your packets somewhere other than along the spanning tree,

because that might cause a black hole. You might send it somebody who doesn't know where they go. So in IP, at every node, you have to have perfect information. You can't approximate it. In CCN the semantics of a FIB entry is this node might have information about the prefix, or information that matches its prefix might be out that face. It's approximate to begin with. And it can be multiple faces. And you can take all of that state and turn it into Summarized Approximation State, as long as the errors that you're making are false positives. You can send the data anywhere, and that just wastes bandwidth, as long as one of the places that you send it is the place that you should have sent it. So in specific, you want to code your routing tables as a Bloom filter, it works perfectly, because bloom filters only have false positives; they don't have false negatives. You're never going to fail to send the data where it should go, but you may, because you compress the state, send it some places where it shouldn't go. That's okay, it'll get discarded, because they don't have the data. And you get that out of the multipoint model, sort of for free, as a consequence of having multipoint delivery. But it means that you can make some – if you care about state and state concentration -- you can do explicit state versus bandwidth tradeoffs. You can say, 'oh, I can compress the bejesus out of the state.' And all that happens is some fraction of the packets are going to get flooded; they are going to go places where they shouldn't and get discarded.

Lastly, the transport protocol used by routing is always a flooding diffusion-based protocol; it has to be, because it's the protocol used to create topology, so it can't rest on topology. Otherwise, there would be a recursion problem there. You have to make routing out of something that doesn't depend on routing. And flooding is all we know that doesn't depend on routing. You can decide it locally. You can implement it locally. It only depends on the information. That means that CCN lets you implement routing protocols directly in the CCN protocol, because its model is what the routing protocols would like to use. So if you want to try any of those things, I'd suggest, hey, do it in CCN. It's just a great discover vehicle. And if you do it, you get infrastructure protection for free, because all of the information secured. It's all signed.

I won't talk about any of that. I want to talk about why/how the world should deploy CCN right now. And you can do that, because it is a universal middleware model – you want to deploy it over the existing IP routing – with longest-match lookups. It's semantically compatible; it expects the same sort of things that IP expects. If you can do that, if you can dump CCN information into normal Internet routing, then you really can deploy it today. And if you can do that, it might give you some intuitions on scaling. It really is just the same as IP routing. It should have the same scaling properties. So in order to describe that, I need to describe how routing works, because it's black art to many people. Probably not to you, but I will describe it anyway. I'm just describing link state routing, because the other one doesn't really work…

In link state routing, nodes announce their existence via a local broadcast, across every link they've got. They send out a hello that just says, "I exist." B sends out of each of its links a hello, "I'm B." A node hears that and learns that it's adjacent to B; B can talk to it. When C hears B's hello – which is local communication on that one link – it floods via a link state announcement what's called an adjacency. It tells the whole world that there's a unit directional path from B to C. The same thing happens in the other direction. C sends a hello; B floods an adjacency that the whole world learns that there's a unit directional path in the other way – C to B.

As soon as the world learns that there's a path in both directions – this is flooded, so it goes to everybody. When everybody learns that there is a path from C to B and from B to C, they make a map of a network that includes a link from C to B. ISIS and OSPF require bidirectional connectivity. You have to hear an announcement from each direction. And that causes you to put a link in your map. Every node in link state routing is building a map of the network. So everybody's playing this game. Pretty soon you're going to get a map of all the connectivity. So that's one part of link state routing. It's the internal part. Each node is creating a map of the entire
topology, and it's doing it via these flooded LSAs, which in turn are coming out of the local hellos, which provide the adjacencies.

So everybody has this map. The other part is using it. Some external agent, something not part of the cloud, says, "I'm adjacent to B, and anything that's prefix 192.168, I can reach it." There's some outside world entity, another routing protocol in the same box**,** BGP or another version of IGP. It supplies a prefix that says, "At this node, that prefix is reachable." B takes that prefix, and it floods it throughout its network, saying, "The prefix 192.168 is reachable via B. You can send to me if you want to get to that." That's *using* the map. It's flooded; everybody learns it.

The prefix announcements aren't protocol-specific, and they're not topology-specific. They're information that you want attached to nodes. It can be an IP4 prefix or an IP6 prefix. It could be a time of day. It could be an MPLS label. It really doesn't matter. It's just some information that you want associated with a point in the topology. In the process of moving the protocols from IP4 to IP6, and then trying to put in MPLS traffic engineering extensions, both OSPF and ISIS really generalize the mechanism for what you can announced at a node. Because they wanted to announce lots of different things: prefixes in multiple protocol families, characteristics of links. They can be encoded in a TLV, a type-length value. You get to choose the types arbitrarily. You say how many bytes of data there is. A code for the type, and how many bytes of opaque data. That can be flooded throughout the entire topology. It's attached to a particular node that originates it.

So, the model is: There are CCN content repositories that have certain prefixes that they handle, which they learn about that via internal CCN protocols called registrations. There is a registration agent that talks to a border router and says, "These content labels are available next to you." Adjacent to B is some set of name prefixes, CCN prefixes. B takes those prefixes and floods them throughout the topology via the IGP flooding mechanism, the TLV mechanism.

Routers that "don't speak CCN" – the guys that are only single line – discard the information. They say, "It doesn't apply to me." And the guys that do speak CCN, they use it to build a FIB. A makes an announcement about the prefixes it handles. This is the FIB built from those announcements. B makes an announcement about the prefixes it handles. It makes thhat FIB entry, and it doesn't matter how you learn it. You're getting the labels bound to points in the topology, via the IGP flooding mechanism. You're using it as a transport. You construct a FIB, because you know the topology and you know where the labels are. You find the closest CCN-capable entity in the direction of the prefix. That's what you put in the FIB.

Here, the closest CCN-capable entities are the originators, A and B. If you upgrade C so it speaks CCN, all those FIB entries turn into Cs. The way to get to A or B via an CCN-capable path is via C. You can dump CCN prefixes because they're semantically compatible with IP and

because the routing protocols are sufficiently general. You can jump them straight into your existing IGPs. You can dump them straight into BGP the same way. It has the same TLV mechanism for the same reason. It's unfortunately not a link-state protocol. It's a modified distance factor protocol. It tends to hide a lot of information, so it may not be best to do it that way. In CCN, the DNS is currently being done in order to handle the inter-domain case.

24. **Architectural Issues with IP Routing**

There are problems with the IP-routing model that don't exist with the CCN model. To summarize the data, we'll reduced to just the node link diagram**:** These were the subnets, the net, the four nets. That's the connectivity. The problem is that it's got a lot of double edges and a lot of loops. IP can't deal with that. You have to impose a topology on that. After the routing computation, there's a spanning tree, so a bunch of the edges have to vanish, even though that's capacity if we could send bits over, we can't use it.

This is a topology that has to span the whole world, and there has to be a lot of consistency in that topology. There are certain levels of churn where you can't send data if the topology is still converging. Core parts of the topology, if they change, it can take a long time to re-converge, to reestablish that topology. There are lots of times when you may not be able to send data, and those states can be triggered by very small things.

A couple of years ago, a really tiny ISDN ISP in India misconfigured their prefixes, and they were injecting via BGP 100,000 announcements and withdrawals an hour. That caused BGP boxes all over the world to fall over. Cisco (boxes) just couldn't handle that rate. TCP could move the bits, but the BGP, and the mesh topology supplied by BGP just couldn't handle that traffic. The BGP router fell over, then you lose the high level, the Tier-1 adjacencies. This churn at the far edge of the network is almost inconsequential. It turns into a churn at the core of the network, and it took down the internet. There were large parts of the Pacific Rim that weren't reachable, and they weren't reachable for a couple of days. Cisco sent out an emergency patch. People were filtering routes, identifying who was doing this. They were filtering out the traffic, but they had to do that manually.

Because this global topology is required to communicate, there are global effects for local actions. It doesn't have to be a big player's local action. It could be anybody's local action. There are global constraints on the forwarding because you have to be on the spanning tree. The intermediate guys don't get a choice, because in IP there's only one path between any two points. When you get a packet at a non-local destination address, you have to forward it. It doesn't matter if you look at it and say, "I don't want to handle this. That's not my customer," "I don't trust this packet," or "I'm really at my quota." You're on the path. If you have it, somebody thought you were on the *only* way to get from point A to B. If you throw the packet away, then you've made a black hole. Those two nodes can't communicate. Because you have to be part of this global forwarding structure, the single spanning tree, you have global constraints on your local routing decision. (IP weakness)

25. **No Topology Needed**

CCN doesn't need topology. At best, it's a hint. It saves some bandwidth. You can use any or all links can be used any or all the time. You don't need topology because the content model suppresses duplicates. There is nothing that can loop. Data gets extinguished immediately

if it goes around a loop. There's something called a *nonce* that is an identifier that goes in the Interest. If they go around in a loop, they get extinguished.

The data bandwidth is going to be a theoretical min. You get the data from the closest point. If you do that by flooding Interest, there's going to be some off-path, off closest point copies that get sent to you and get extinguished because they're duplicates. If you have topology, then you don't send those off-path Interests, the amount of win that you get from doing that is a really small factor. It's a fraction of the average degree times the diameter of the Net. Those copies would tend to go off and then remerge again unless the data's really in disjoint places. What you win is those little splits. If you actually compute it, it's a tiny number.

You actually get CCN to work best if it's got crummy or approximate topology. You do something that we call *performance-based Interest re-expression*.

26. **Transport State**

There's this tree-based content model, you're asking for things an item at a time. How do you get big stuff? How do you say what you want? That's the fundamental problem of transport. It's easy in a stream-oriented transport, because basically you're enumerating a byte stream. And a line is totally ordered. If you can say what you've got. You say that mark between what you've got, that's what you want is everything after that mark. That's what TCP does. That's what a TCP sequence number is. There's some point that an ACK identifies, that's everything that you've got. Everything after the ACK, all the bytes higher in sequence order, is stuff that you want. And there's another parameter, a window – that's how much that's allowed to be in transit, how many of those bytes the sender is allowed to send. That's a really compact state. It's really easy to robustly move data.

To do this same thing with this tree-structured CCN namespace, you take advantage of the fact that the one data structure in addition to a line that you can totally order is a tree. You can't totally order a graph. You can't totally order any two-dimensional or greater structure, but you can totally order a tree – in fact many different ways. We ordered and preordered, lexographic order, and that means that these names have a total ordering, therefore I can say what I want relative to what I have. That specifies a particular path through the tree. And I put the conventions down here. The tree is lexically ordered. If you don't give a relationship, what you're asking for is the leftmost child going down the tree. So you're going to follow the branch.

A lecturer decides on a name – "Slides from a talk" – and wants to ask for the highest version but doesn't know what the highest version is, he can ask for the base name of the talk, not including the version number. He asks for the rightmost child of that basename. Version numbers are ascending integers, so the rightmost child is the highest version. That's the application convention. Asking for the rightmost child provides the most recent version, and the lecturer doesn't need to know what it is. And then from the rightmost child, the default convention is follow down the leftmost child. Its leftmost child is block zero in that piece of content. That name is going to provide the first block of the most recent version. I don't have to know anything more than the basename.

If the lecturer wants the next version, it pops up. To find out if a new version has been created, he asks for the version after version 2. It's the rightmost sibling of version 2. There isn't one. And he goes back. If he talks to something that is authoritative or repository for that content, it's going to give back a NACK that says, "I know, and it doesn't exist yet." If he doesn't get to

an authoritative repository, it's going to time out. If he has chunk S1 and before, and wants some chunk after it, he asks for that name. He gives the name of the most recent thing he got, and then implicitly, because of the resolution rules, is the next thing, anything afterwards.

So the names are what you use for transport state. Basically, you get a packet in the default case, and you turn the name in that packet into your next Interest. That Interest is asking for the thing after what you've got.

27. **Content Storage and Travels**

There are nearly 10 million people that read the *New York Times* online in the US. They all pull a copy from NYTimes.com. It's not Akamai'ed because NYTimes.com doesn't want to tell its advertisers exactly how many people read it. How do you do that with CCN? The model is sort of a recursive model. A client announces its Interest. The Interest percolates back to NYTimes.com, because nobody along the path has it. NYTimes.com sends the data. As it goes back down to the client, it's remembered in the content store of all of the routers along that branch. The data are delivered to the client, but everybody who is marked in blue (chart 34:23) has a copy. Client 2 asks for the same data, so its Interest propagates up. As soon as it hits a place where the content exists, the Interest matches the Content Store, and it's sent from that point. Basically, you add the branch from the client to where the data already is. You're creating a CDN dynamically, specialized to that particular set of Interests.

You can prove that any piece of data is going to take at most one trip across any given link. After it's taken a trip, it's in the downstream content store. It's going to be satisfied out of the content store, so it doesn't need to be pulled across the link again. That's optimal. You can't do a delivery network that uses less bandwidth than that. Because in this model and most of the models you're distributing along a source tree, i.e., there's a root that has content, and it's going out to all the consumers of that content. In general, all of the mass of a tree is at its leaves. (35:41) It doesn't have to be a binary tree. That means there are a lot of copies that get seated very close to the consumers, which means that the average latency – how far you have to go to get your copy – is very low, because the data very rapidly percolates from the root of the tree deeper into it, which means it is closer to the consumers.

None of these replicas announce the replicas. We're putting the data in the content store, but to be useful, it doesn't require that we say via routing that we've got the data. The way that the data is found is because we're on the path to the source of the data, it's discovered from the routing that defines the distribution tree. This is the same model that is used in things like Core Based Trees Multicast. That is, you don't have to put state everywhere. You can have a very high-level state, things that define repositories. That induces trees, which can then be used to reduce the distribution of replicas, and let you use those replicas efficiently: minimum bandwidth; mnimum latency; minimum link use. So you make a close to optimal CDN, with no control traffic automatically, and it's specialized because it's done on each particular data item. The distribution network is specialized per those data items.

28. **A Comparison of the bulk-transport performance of CCN versus TCP**

It's not just a request/response protocol. you can embed TCP/IP in it. You would like it to have the same performance. Here, we're varying the window size for TCP. That's the x-axis, the pipeline size, and we're verifying the number of simultaneous Interests you could have in

process for CCN, which is the same parameter. How many unsatisfied Interests you can have, outstanding. We're varying it from 1 to about 50. The numbers there are packets to make them directly comparable. You can see for TCP it runs stop and wait when it's only got one packet. It increases linearly as you give it more packets. This is a local test, so there's no transit delay. It's all store-and-forward delays, which is why it goes up so steeply and why it's really flat, not round. Then we saturate at the link bandwidth, which is 100 MB per second.

This is TCP - a kernel-level implementation which has been continuously tuned for twenty years. It's pretty good. CCN is a task-level implementation, which has been running for a couple of weeks. It's pretty bad. The leading edge is pretty grim there. It's a big pipe – because it's going from application into the kernel, out to another application, back into the kernel. You do that two times per hop. A lot more store-and-forward delays, a lot more rise time, but it goes up, and it flattens out. We get an asymptotic throughput. From the asymptotes, you can read the relative efficiency of these protocols . TCP app bandwidth, 11% down from the link bandwidth. That's the size of the headers – the Ethernet header and the 40 byte TCP/IP header. The CCN is encapsulated in UDP, so it has the same Ethernet and IP headers on it plus an CCN header. It can't do any better than TCP is doing, and does 23% worse. That's basically the size of the name and the size of the SHA256. The name is as big as a TCP header. and the SHA256 is as big as a TCP header. There is exactly twice the overhead. Within the overhead, there is the same asymptotic throughput. We saturate the wire. We're running flat. At the wire, it's running 100 MB per second. It's down 20% in the throughput because it's not an apples to apples comparison. CCN does security, and TCP doesn't. Basically, the factor-of-two difference is the security overhead.

Things change a lot if you look at a sharing test, so here we're distributing a Youtube video to four clients via TCP and via CCN.over a 10 MB Ethernet, so it's a slow bottleneck link. In TCP, each client posts on a separate copy, so as you add more clients, this is measuring the time until the last guy gets the video, so it's the time to completion for all four. For TCP, the time to completion goes up linearly as you add the clients, because they're all sharing the same bandwidth, so as you add a new client, you're halving the bandwidth for each of the others. For CCN, the number of clients doesn't matter, because they all share the same broadcast protocols. They all share the same copy. So irrespective of the number of copies, you get the same performance.

29. **Mobility Management**

In CCN, issues of mobility management are much simpler. They are a reality in the modern world and hard to deal with. But they are way harder to deal with using IP. The core one, in IP, is who is being talked to matters. When that changes, the fact that it's changing has to be pasted over – using mobile IP or some other tunneling mechanism. You have to bind a host identity. In CCN, it doesn't matter who you're talking to; it matters who you're talking about. So if who you're talking to changes, you don't care. You didn't care to begin with. You don't know! So there's a level of mobility management, the session preservation, that simply doesn't exist, because you're dealing at the content level, and that's immune to mobility. When I move from here to there, I still want the same stuff, so I just can keep asking.

There's another issue of mobility management that is connection-orientedYou ask for, a YouTube video, and you get the whole thing, because the segmentation is a transport-level abstraction, not a network-level abstraction, so getting data is all or nothing.

So when you're pulling down a YouTube and you move to a different place, the data that come towards you have to somehow re-vector to that different place. And you don't want to waste a lot of bandwidth putting it in the wrong place. It constrains the way the that you construct tunnels and the way that you dynamically want to construct tunnels, and you have to predict where you're going to be, in order to correctly move the data, and not send too much of it to the wrong place. It's because you're dealing with very large units. In CCN, the segmentation happens at the lowest level, and you ask for a packet at a time, so it doesn't matter if the original piece of content was 60 GB; it's being received 1KB at a time. And if you're talking to something upstream that says, "We have a 2 packet link-bandwidth X delay product. You give me a thousand outstanding Interests, I'll forward two, because we have a 2 packet link-bandwidth X delay product, and you only need two to fill the pipe."

So you don't have a lot of data coming at you. And if you move, you haven't made a very big mistake. And because that data is being cached all along the path, even if it goes to a place that you've been, it also went to the guy upstream of him, and that's a copy that you can get in the new place that you are. You can take advantage of the locality, which means that you can do very efficient mobile distribution being very stupid. You don't have to predict where you are. You don't have to know where you're going. You don't have to know where the infrastructure is. Because you are not making a bunch of big mistakes that you then need to fix.

And lastly, because looping isn't an issue, all the links can be used simultaneously. If you're trying to figure out which link works to get a piece of data, the answer is to try them all – out of a set of carrier wireless, infrastructure wireless, point-to-point connections, bluetooth. Something is going to respond, and Interest and data are always paired. There's no uni-directional communication in CCN. You can experiment, conditional on policy, and measure results, and change future behavior based on measured results.

30. **A New Layering**

This is the CCN protocol stack. It takes the IP hourglass. The waist, IP, that narrow waist in the stack, has been tremendously valuable. It let us evolve what's below it and above it, in many different dimensions – new services, new applications, creations like the Web, like VoIP. But there may be other things that are equally as valuable for the waist – little chunks of content, CCN content, signed named pieces of data, that would be a really appropriate waist for the world as it is today. You need to put two new layers around that. The first is the *Strategy Layer*, which is basically the mobility management layer. It has to be in the stack, because in the world today, there are lots of ways to communicate. And you need to figure out dynamically, almost on a packet by packet basis, what's the best way to communicate. That needs to be in the stack, and the stack needs to be engineered, so you can use that information. The second is a security layer. Trust is too important. Provenance is too important. The network of communication is too much a part of our lives. And that has to be in the stack, and, at least, the identity part can't be optional. Signing has to be possible. Other than that, it looks pretty much like an IP stack. It's as simple and provides roughly the same performance.

[i] Published by Andy Odlyzko, MINTS program website, University of Minnesota.

[ii] IDC study, 2008.

[iii] This is data from Thompson at IBM. He did a survey article in IBM Journal of Research and Development about the dollars per byte of IBM disks at the time that they were introduced. He looked over 20 years. I graphed 10 years of that data, and that's a log-linear scale, that's an expediential decrease.

There's another set of data that comes from Doug Galbi, who was Chief Economist at the US FCC until Bush seized power. And he was surveying carrier prices. When Bush took power, all that data vanished. I picked one – OC-3 prices over the same time period. That's this axis; it's also a log axis, but it doesn't need to be. And it's for an OC-3; it's dollars per Megabytes per mile. One is an expediential decrease; the other is flat. And I generally get grief about this graph because we all know that communications costs have been going down. They haven't been going down anything like that fast. There's actually useful data here about how telcos work.

[iv] in 2008, the Max Planch Institute showed

[v] This month, University of Twente announced

[vi] LBL

[vii] LBL

[viii] So for many years – for decades – both John Backus of IBM and Tony Hoare, now of Microsoft Research, got the ACM Turing Awards for preaching against containers.

[ix] Ron Rivas

[x] there's a whole class at MIT on lock-picking.

[xi] and this is another idea from Ron Rivest. It came from a system that he did called 'SDSI' which I'm going to talk about in a little bit.

[xii] there was a thing invented by Ralph Merkle

[xiii] Ron Rivest, who is one of the true gods of security, is the 'R' in RSA, and invented just an incredible amount of fundamental security infrastructure – he got very frustrated by the state of certification in the 1990s, and he invented. (Simple Distributive Security Architecture)

[xiv] He published this in '96. He took it into the IETF, tried to get it standardized, and it turns out that was a very frustrating experience.,

[xv] In the slides, these are links – that's to Ron's SDSI page, the axiomazation was done by Joe Halpern, and that's a link to his paper, which was done in '99. That's the trust model we use. It is very good, and nobody can match Rivas for designing these kinds of things.

[xvi] Now, these are my words, not Diana. … John Maynard Smith, the evolutionary biologist who gave us evolutionary gain theory and a lot of mathematical insight on how ecosystems and organisms create themselves – how they self-organize and structure – in one of his books, he said that

[xvii] but if you look at Rivest's webpage, a lot of their work has been on signing,

[xviii] Most of Ron's recent research has been on micropayment systems and on small-transaction systems where the users want anonymity.

[xix] It really boggles the mind. There was a paper in the last talk from Martin Dietzfelbinger (4:43) of University of Illinois and several colleagues, Well, if you looked, at the theoretical computer science conferences over the last decade.

[xx] You couldn't do that – Don Knuth said you couldn't do that,

[xxi] But between Rasmus Pagh at University of Copenhagen and Martin,

[xxii] there's a beautiful piece of work by Frank Kelly of Cambridge – a book he published in '79 – which I didn't read it until '89. Would have saved me a hell of a lot of time if I had gotten to it a decade earlier. Called 'Reversibility and Stochastic Networks.' It talks about the properties of this detail balance.

[xxiii] and so the same reason that when Cisco did PGM – their multi-cast transport – they had to stick part of it into the routers, it's because the data needed to be replicated.