computers are bad

2021-11-26 no u pnp

Previously on Deep Space Nine, we discussed (read: I complained about) the short life and quick death of DLNA. On the way, I mentioned DLNA's dependence on UPnP as an underlying autoconfiguration protocol. Let's talk a bit more about UPnP, because it's 1) an interesting protocol, 2) widely misunderstood, and 3) relates to an interesting broader story about Microsoft and Web Services. It'll also be useful background knowledge for a future post about the thing I originally intended the last post to be about, the Windows Media Center-based home automation platform exhibited at Disney's Innoventions Dream Home somewhat over a decade ago.

UPnP has sort of a bad reputation, and it's very common in online conversations to see people repeating "disable UPnP" as stock advice. There is some reason for this, although the concern is mostly misguided. It's still amusing, though, as it relates to the one tiny corner of UPnP functionality which has ever really had any success.

Before we dig into that, though, we need some history.

UPnP fits into a broader space called zero-configuration networking, which I will be calling 0CN for short because the more common Zeroconf is, technically, the name of a specific protocol stack.

0CN aimed to solve a big problem for early adopters of local area networks. You can connect all your computers to a network, but then they need to be numbered (addresses assigned) and then configured to know what addresses are capable of offering which services. In early small networks this could be relatively simple, but between the mass migration onto the relatively complex TCP/IP stack, the ubiquity of "non-consumer" automatic address assignment via DHCP [1], and the increasing number of devices on home networks caused by WiFi... it was turning into a real pain.

It's just hard to sell a product to a consumer as "fun and easy to use" when the setup instructions are going to involve some awkward arrangement to get the device's IP address, then having to go enter that into other devices in order to use the new service. But with a typical pre-0CN home network environment of IP addressing assigned by DHCP (probably by ISC dhcpd running on a consumer router), this was pretty much the only option for devices or software to discover and connect to other network participants.

Some network developers had addressed these problems surprisingly thoroughly in various pre-IP network stacks. For example, AppleTalk is often raised as a prime example--AppleTalk is a pre-IP protocol based on Xerox's XNS that had autoconfiguration and service discovery baked into it from the start. A number of other early LAN protocols were similar. But the "everything over IP" trend could be analogized to parking a heavy diesel truck in the driveway. It's highly capable, which is attractive, but it was not designed for consumer use. The industry loved TCP/IP as a powerful common platform, but typical home computer users didn't want to learn to double-clutch. In this way, TCP/IP for home networks sometimes felt like a regression compared to the various (often XNS-based) ease-of-use-oriented precursors.

0CN, then, can be viewed as an effort to *re-consumerize consumer networks,* by smoothing over the parts of TCP/IP that were not easy to use.

0CN systems thus tend to concern themselves first with discovery, that is, the auto-detection of services available on a local network. Discovery tends to further imply description, which is the ability of discovered devices to describe what they are capable of and how to interact with them. Most computer interconnects have some form of discovery and description, but 0CN differs in that these capabilities are intended to be very high-level. What is *discovered* is something like a media server, and it *describes* itself in terms of protocols that can be used to retrieve media and their endpoints. This is a much more end-use oriented form of discovery than we see in lower-level discovery protocols such as ARP.

0CN systems may, or may not, extend discovery and description with a set of standardized communication protocols to be used after discovery. For example, 0CN standards may specify the type of API that a device should present once discovered. This may include standardized message bus protocols and other application-level concerns. Later 0CN systems tended to be more prescriptive about this kind of thing to ease implementation of "universal" clients, but there are still plenty of 0CN standards around that leave all of it to individual device and software vendors.

One of the weird things about 0CN is the relative *lack* of standardization. It's not that there isn't a well-accepted standard for 0CN, it's that there's like four of them. To be reliably discovered by a variety of operating systems, devices typically need to implement multiple 0CN standards in parallel.

Here's a brief summary of 0CN protocols in common use today:

In the Windows world, NetBIOS specifies a basic discovery system for both advertising capabilities and name resolution (of NetBIOS names) [2]. This protocol is looking pretty crufty today but is still in reasonably common use between Windows hosts offering file or printer shares. In part in response to the limitations of the NetBIOS mechanism, Microsoft introduced WS-Discovery (part of Microsoft's larger Web Service craze which we will discuss in the future), which is a little more modern and a little more powerful. It is still frequently used by network printers to advertise their capabilities. UPnP, as a Microsoft protocol, is widely supported by Windows hosts and less widely by devices such as printers and consumer NAS.

In the closer to UNIX world, discovery approaches derived from DNS are popular. The most prominent is a combination of the mDNS distributed DNS service with the DNS Service Discovery (DNS-SD) standard, which allows the distributed DNS mechanism to be used to describe capabilities as well as presence and name (basically by using a set of specially crafted SRV records). Apple's Bonjour and the open source Avahi are both implementations of this mDNS-and-DNS-SD combination. "Zeroconf," with a capital Z, has a tendency to refer to this stack as well but there's a lot of inconsistency in how the term is used.

So this is the landscape in which UPnP exists--already somewhere between solved and hopelessly fragmented. UPnP intended to compete, in part, by being a more complete standard with higher level components. So, for example, UPnP specifies the general structure of device APIs (XML SOAP as discovered from the XML device description), an event system (basically a simple subscription message bus), and a very lazy end-user standard that basically suggests that all UPnP devices should offer a web interface.

More significantly, UPnP was expanded significantly into the A/V use-case. The UPnP A/V spec includes a basic media architecture, including media servers, renderers, and control points. UPnP specifies not just discovery but also protocols between these devices. You are likely now wondering what the difference between UPnP and DLNA is, and that can be a confusing point. DLNA is directly based on UPnP, or perhaps it is more accurate to say that DLNA *incorporates* UPnP. DLNA uses all of the UPnP protocols, including A/V, but extends the UPnP specification with much more detailed standards for content management. A simple way to think about it is that DLNA is the upper level while UPnP is the lower level. This of course does not totally help with the fact that DLNA was so closely associated with UPnP during its lifespan that it's not uncommon for people to use the two terms interchangeably.

So, how does UPnP actually work? First, UPnP does not handle addressing but instead incorporates either DHCP or RFC 3927 link-local addressing. So, UPnP functionally starts at the discovery layer. For discovery, UPnP incorporates a protocol that never quite made it on its own: Simple Service Discovery Protocol, or SSDP. SSDP is actually as simple as the name suggests, but due to Microsoft's incredible love of web services it uses an unusual transport. SSDP runs on top of HTTPU, or HTTP over UDP, a protocol that basically consists of taking a small HTTP payload and putting it in a single UDP packet.

A client wishing to discover services sends a simple HTTP request, in a UDP packet, to the multicast address 239.255.255.250, port 1900. Any device that has a service to offer replies with an HTTP response to the source IP and port on the request. To help reduce traffic volumes, SSDP also allows devices to gratuitously advertise their presence to the multicast address, and all clients are permitted (and encouraged) to passively discover devices by monitoring these extra announcements.

The next step is description. UPnP description is quite simple: the device discovery information provides a URL. The client requests that URL to receive an XML document that describes the device and gives a list of services it provides. Each service is defined as a set of endpoints and commands that can be issued to those endpoints.

For the most part, a UPnP client will now interact with the device by issuing commands to the described endpoints. This is all done, following the general trend you may have noticed, with XML SOAP over HTTP. That's basically the end of the UPnP story, but UPnP does add one interesting additional feature: an eventing or message system. The UPnP description of a service lists a set of variables that describe the state of the service, and provide an event endpoint that allows a client to subscribe for notifications whenever a particular variable changes. This is all, once again, done with XML over HTTP.

So this is all interesting and helps to describe the underlying structure of DLNA (which can be viewed as just a set of standardized, strongly specified UPnP services). But it has oddly little to do with what we know UPnP for today. What gives?

The service discovery and description functionality of UPnP is, for the most part, boring and transparent. It is used in practice for things like autoconfiguration of printers, but it doesn't usually do anything that notable. What UPnP is broadly known for is *NAT negotiation.*

Many home devices that offer services might want to be internet-accessible, but the fact that most home networks employ NAT makes it difficult to set that up. UPnP intended to resolve this problem by throwing a NAT port mapping protocol into the UPnP standard. Called Internet Gateway Device Protocol (IGDP), it allows a UPnP client to discover a router, and then specifies a service the router provides that allows a UPnP client to request that a given port be opened from the internet back to that device. As a bonus, it also allows a UPnP client to request the current external IP address from the router, so that it knows its internet endpoint.

IGDP is a sibling to Port Mapping Protocol and the later Port Control Protocol (PCP), but for a few reasons it is much more common. One of the reasons is a business one: it’s pretty much correct to say that UPnP IGDP is the Microsoft solution, while PCP is the Apple solution. PCP is implemented and used by a variety of Apple products, but UPnP support is more common in consumer routers.

In the eyes of many power users today, UPnP’s service discovery function is so little discussed that UPnP is more often used as a mistaken reference to IGDP specifically (router vendors and innumerable support forums do not help to alleviate the confusion here).

Because of security concerns around providing a low-effort way for software to map incoming ports, it’s common security advice to “disable UPnP,” which in practice means disabling IGDP... but since IGDP is the only meaningful service provided by most consumer routers, that’s often done by disabling the whole UPnP implementation and it’s labeled as such in the router’s configuration interface. I am actually somewhat skeptical of the security advantages of disabling UPnP for this purpose. The concern is usually that malware on a machine in the local network will use UPnP to map inbound ports, allowing direct inbound connections to things usually not accessible from the internet. That problem is real, but in practice malware running somewhere on the local network has many options for facilitating external access and UPnP isn’t even really one of the easier ones.

There’s actually a much better and less often discussed reason to be cautious about UPnP: there is a long track record of embedded devices like routers and IoT “things” having poorly implemented UPnP stacks that are vulnerable to remote code execution. Several botnets have spread among IoT devices through UPnP, but it had nothing to do with port mapping... instead, they took advantage of defects in the actual UPnP implementations.

What’s worse is that many IoT UPnP implementations turn out to have a defect where they do not correctly differentiate requests coming from the LAN side vs. the WAN side. This has the alarming result that it is possible to request and receive port mappings from the WAN back to the WAN. All devices with this defect are potential participants in reflected DDoS attacks, and indeed have been widely abused that way.

The point is that I would agree that it’s a good idea to disable UPnP, but not because of *what UPnP does,* and not just on your router. Instead, it’s a good idea to be very skeptical of UPnP because of defective implementations in many embedded devices, especially routers, but also all of your IoT nonsense.

Ultimately, 0CN almost feels obsolete. Most modern products have a near total dependence on a cloud service, and simply use the cloud service as a broker instead of performing service discovery. Where devices do need to be discovered, it’s usually pre-configuration for WiFi networks, and so it has to be done over a side channel like WiFi Direct or Bluetooth. This has the substantial downside that these seemingly local devices will stop working if the internet connection is lost or the service unresponsive, but it’s the 21st century and we’ve all come to accept that our light switches are now dependent on Amazon somehow.

More seriously, there is a general trend that more expensive, higher-quality IoT products are more likely to perform service discovery and communicate on the local network. This likely happens because their developers realize that support for local communication will result in lower latency and better reliability for many users. But ultimately reliance a cloud broker is easier, so a lot of even high-end products ship a cloud intermediary as the only way to communicate with them.

[1] DHCP might sound like it would be part of a 0CN environment, but 0CN is usually used to describe easy-to-use, highly automatic, consumerized protocols *in contrast to* configuration protocols like DHCP that were designed for large networks and so are relatively complex and difficult to work with.

[2] Peer-to-peer discovery and name resolution protocols tend to have inherent scalability problems due to their dependency on broadcasting. NetBIOS, having originally been designed as a pretty complete and sophisticated network application standard on top of pre-IP protocols, resolves this scaling problem by supporting centralized name service for large networks. That centralized name service is called Windows Internet Name Service or WINS, an acronym you have probably run into before dealing with Windows network configuration. WINS is essentially a pre-IP DNS, for the much more limited NetBIOS name standard. It is fairly rare to encounter a WINS server today as Microsoft has shifted to DNS for almost all purposes, even for establishment of NetBIOS connections where the difference between “NetBIOS name” and “DNS name” can now be very fuzzy. E.g., the UNC path “\\a.multipart.domain.name\a\share” has worked fine for many years, even though the name is illegal for NetBIOS... the Windows NetBIOS client has been fine with using DNS names pretty much since NT stopped being called NT. NetBIOS names are also still supported but it’s rare to encounter a machine where the NetBIOS name is not the same as the local part of the DNS name, making it not obvious that the distinction even exists.