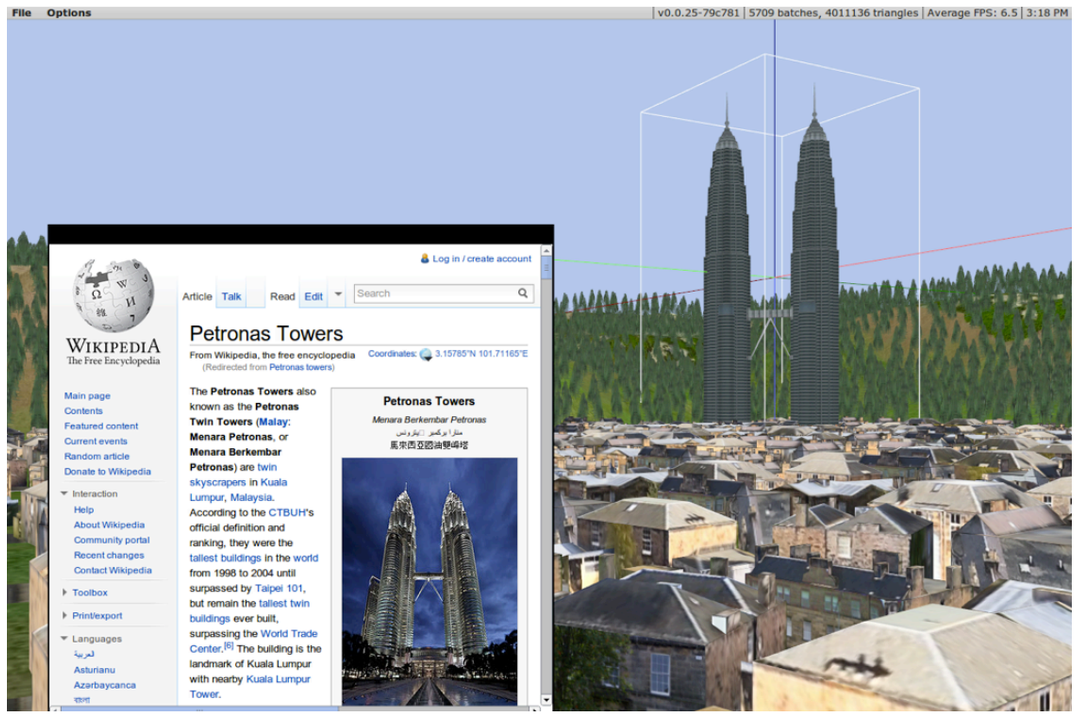
Beyond Porting Sirikata to the Web

Hi everyone, my name is Ewen Cheslack-Postava and I'm a graduate student at Stanford working on scalable virtual worlds.

Sirikata



Seamless, scalable, and federated metaverses

We developed a new platform called Sirikata intended to support a variety of virtual world applications, including metaverses where users control the appearance and behavior of objects in the world. This flexibility provided by metaverses make them very challenging to build, and we started working on a new platform because we believed approaches taken in current systems to at least one challenge, scalability, weren't going to cut it for the type of experience we want to provide.

I've shown a couple of examples of applications built on Sirikata, including a very large world that demonstrates a level of scalability not really achievable in other systems. Besides scalability and other distributed systems challenges, we're also interested in other aspects of the system, for example programming language designs which enable and encourage novices to be creative in the world.

I want to motivate my talk today by zooming in on one of these pictures. Our current client struggles with this scene of only about 60,000 objects, even after the system does a lot of work to automatically combine and simplify meshes. Although this particular scene was constructed with lots of reusable components, the reason the system struggles is because the input is a ton of user-generated meshes and the system needs to figure out how to take the present them to the client in a way that's efficient for rendering and still allow it to interact with individual objects. Scaling display and interaction with this dynamic input is challenging.

Virtual Worlds in the Web

- Much experience transfers to the Web
  - Back-end system design enables efficient, web-based front end
- Constraints of the Web lead to more refined system decomposition
- Deeper integration with the Web

Moving to the web, we'll encounter similar challenges with these environments, but frequently to a greater degree. I hope to make 3 points today about how we can successfully build virtual worlds in the web. First, much of the experience we have building these systems natively will mostly transfer, but backend systems need to be updated to better support the constrained browser environment. Scene graphs, for example, look basically the same on the web, but rely on the backend to provide data in the right format. Unfortunately only system designs for a few narrow domains, such as games, are well understood.

Second, the additional constraints of the web will lead to a different, and in our case, more refined system decomposition. Our port to the Web was fairly direct since it was very experimental, but in retrospect, porting the entire component we did was probably a mistake. Some of its tasks aren't things the browser should really be doing. A more careful system decomposition could lead to a more efficient solution, both in terms of system resources and programmer resources.

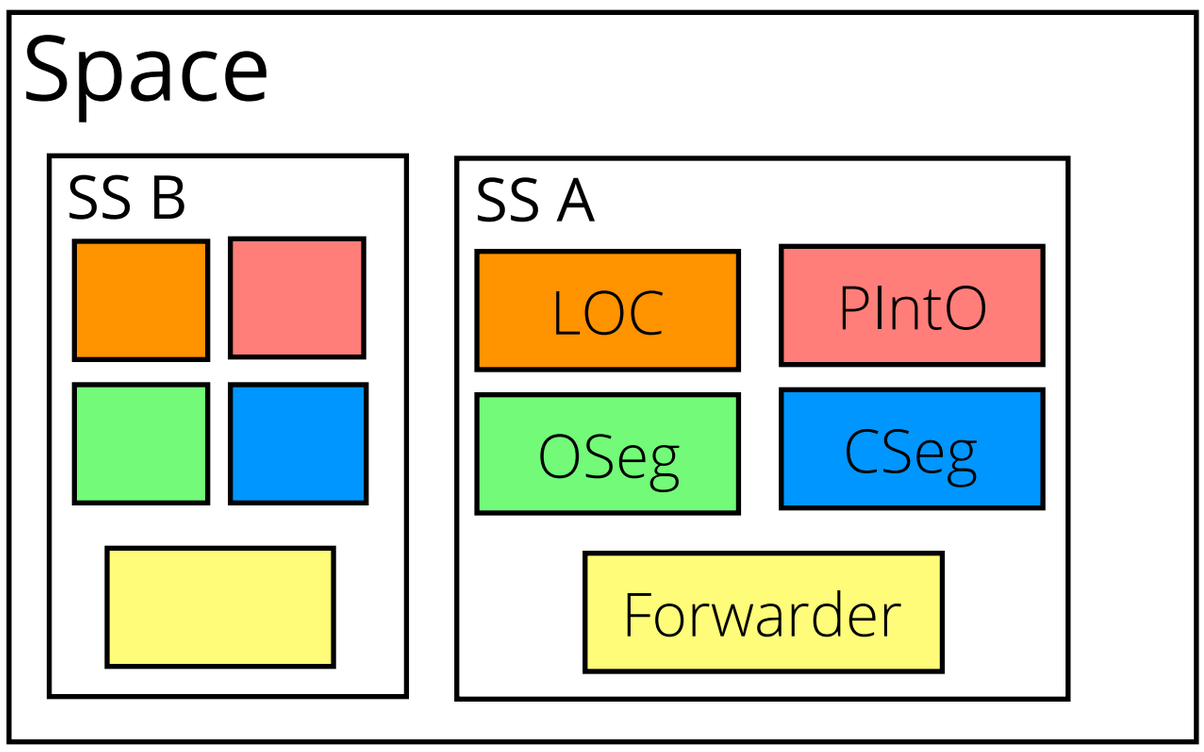
Finally, I think of "virtual worlds in the Web" as much more than just having a client in the browser. That's an important, and very challenging, first step, but we can gain a lot more through deeper integration with the web. True integration with the web, beyond just a web-based client, will allow for a degree of interaction with other systems and composability virtual worlds haven't enjoyed yet.

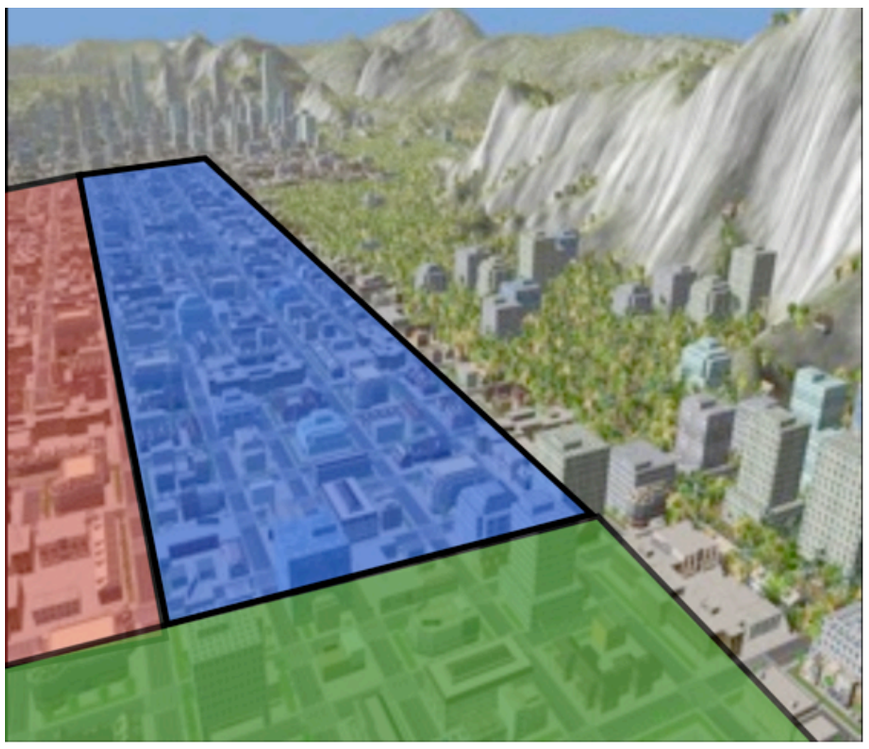
Outline

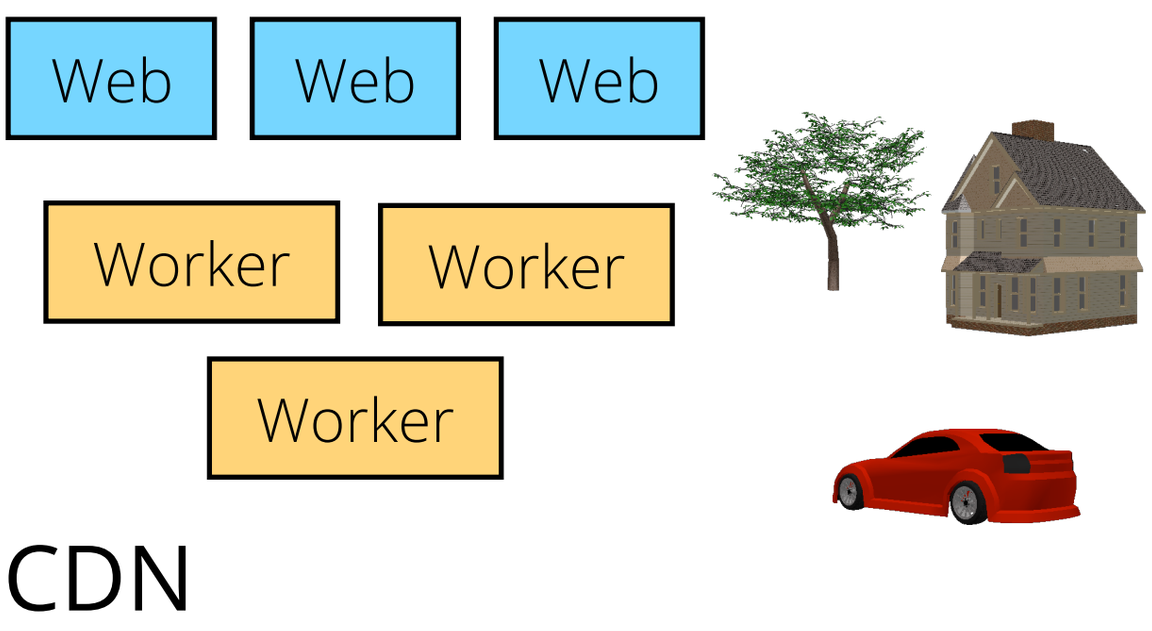
- Sirikata Architecture
- Porting to the Web
- Lessons Learned
- Evolving the Architecture

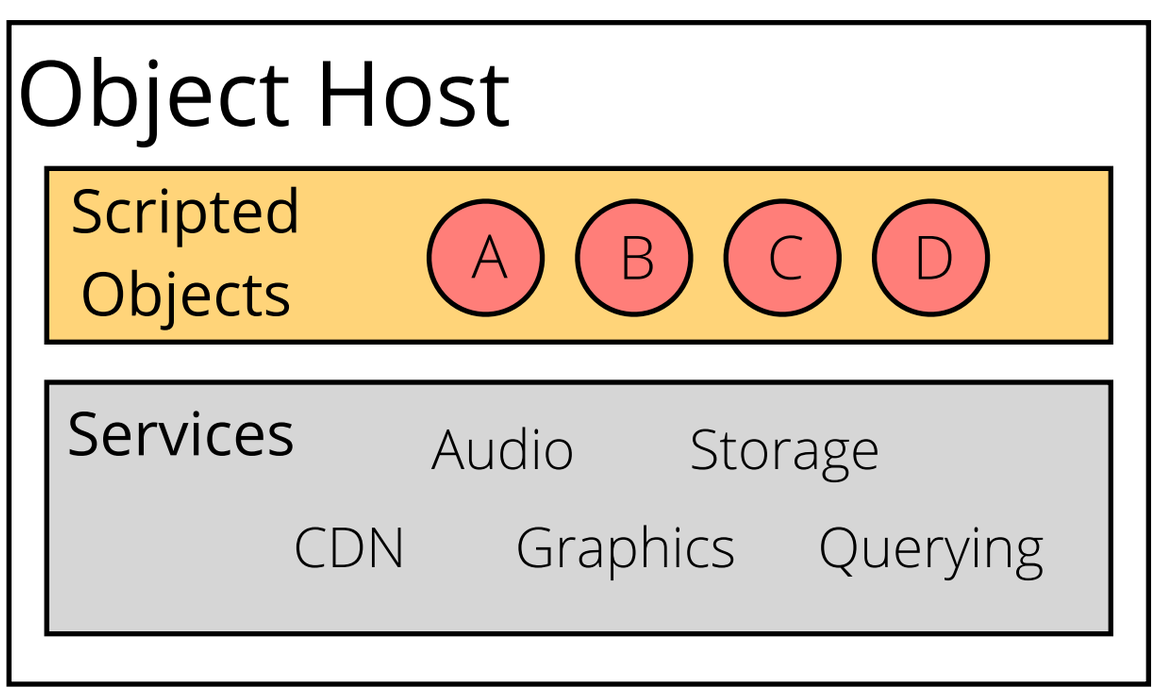
For the rest of the talk I'm going to briefly describe Sirikata's architecture, then explain what we ported to the web and how we did it. Then I'll talk about the lessons we learned from the experience and how they are influencing our thinking about Sirikata's architecture.

Sirikata Architecture





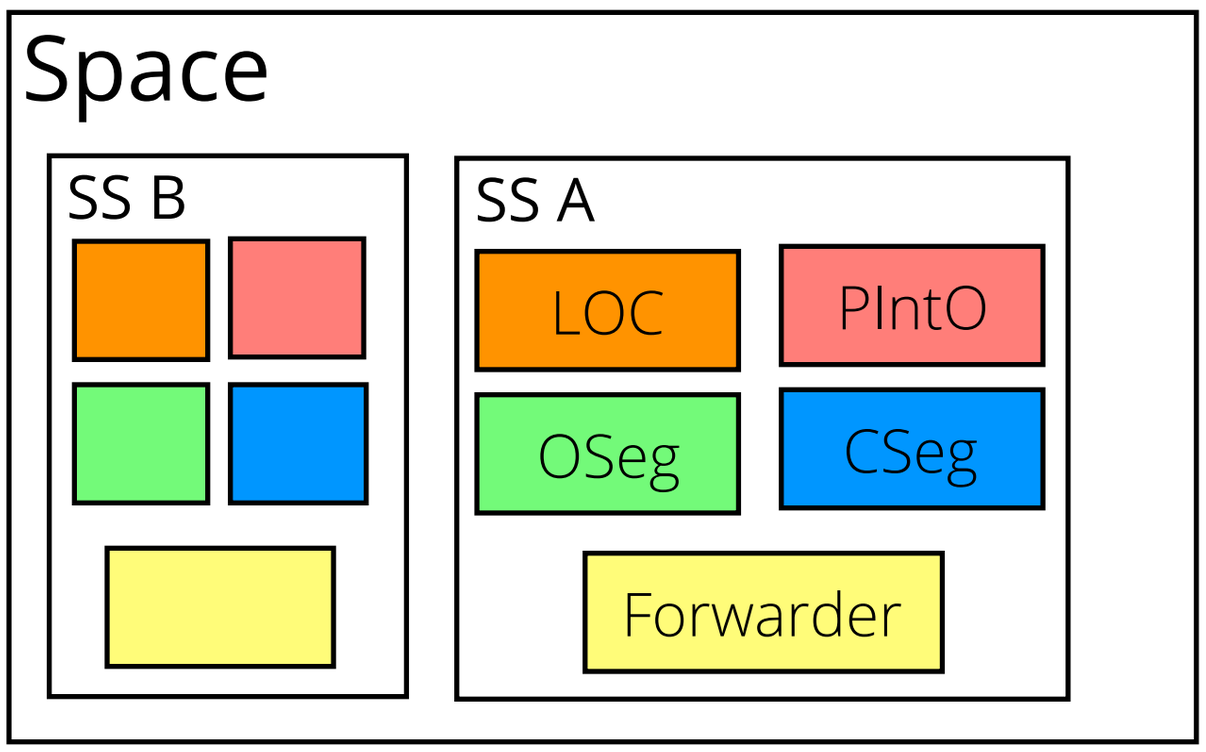


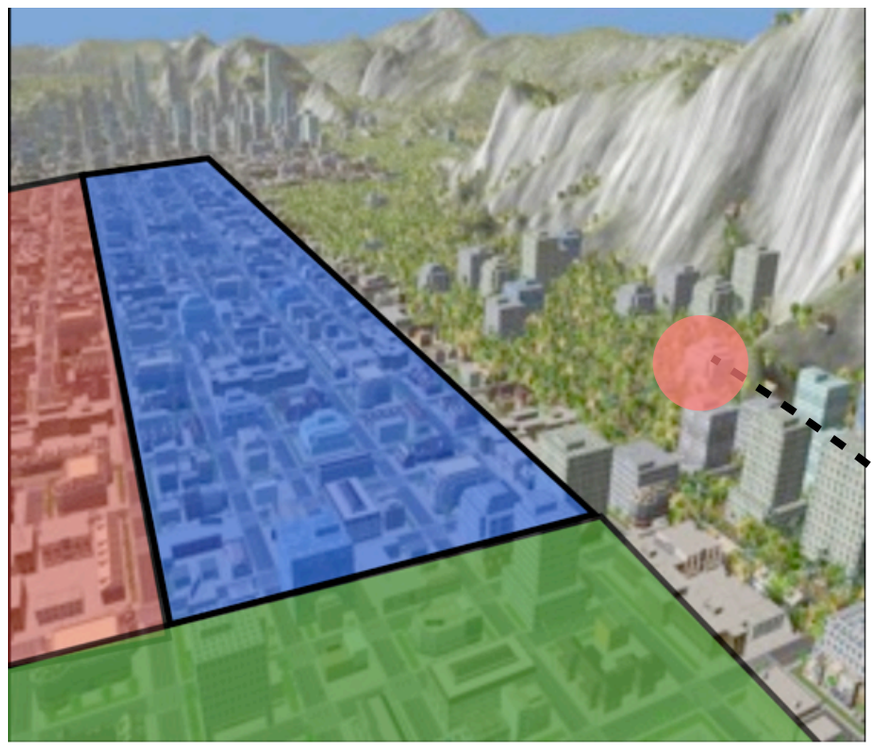


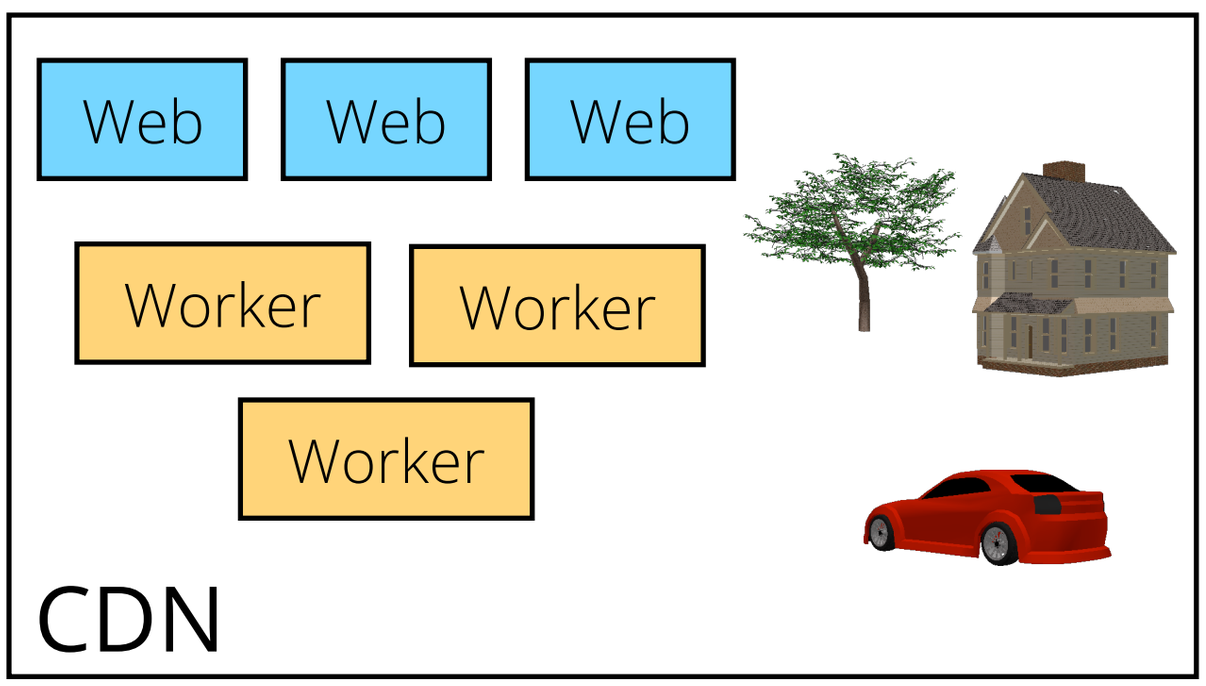
At a high level, the Sirikata architecture is straightforward and should look familiar if you've ever built a virtual world application. The object host, multiple of which can connect, provide basic object simulation by running object scripts. They connect to the space in order to interact with each other. The space tracks their basic properties like location and mesh, provides querying to let objects discover each other, and message forwarding for interaction. The space is broken down by another service, CSeg. Each region is managed by a space server. Finally, the CDN, or content distribution network, manages large static content like meshes or audio clips. The name is now a bit of a misnomer: besides distributing this data through HTTP frontend servers, it also accepts uploads, and optimizes them for progressive download and display. At a high level, you can think of these three components provide 3 abstractions: the object host provides computation, the space provides communication, and the CDN provides storage.

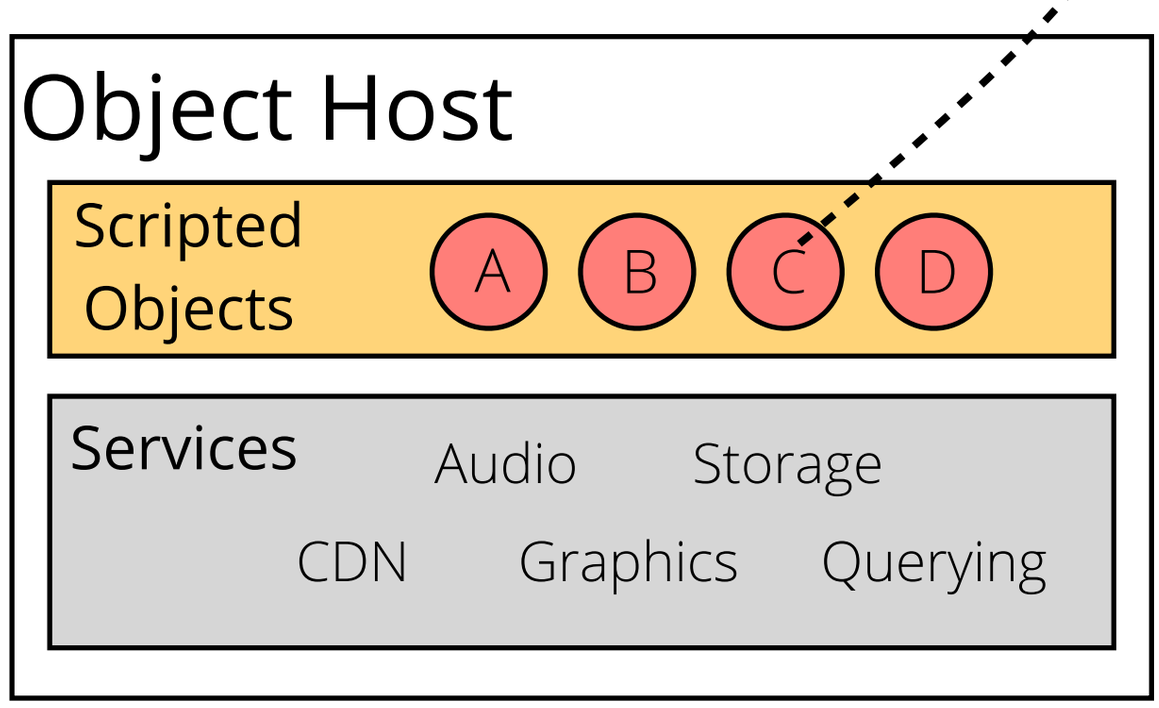
As a quick example of how these components interact, consider an object on an object host. When it's loaded it has no physical representation. If it wants to add itself to the world as a tree, it would initiate session with the appropriate space server and request the location and mesh. This would be added to the space servers *location service*. The location service would store a URL reference to the mesh on the CDN. If an avatar, which is just an object, views the world, it will query *PIntO* for *potentially interesting objects*. When it gets the tree object, it uses the URL to go to the CDN and download the mesh.

Sirikata Architecture





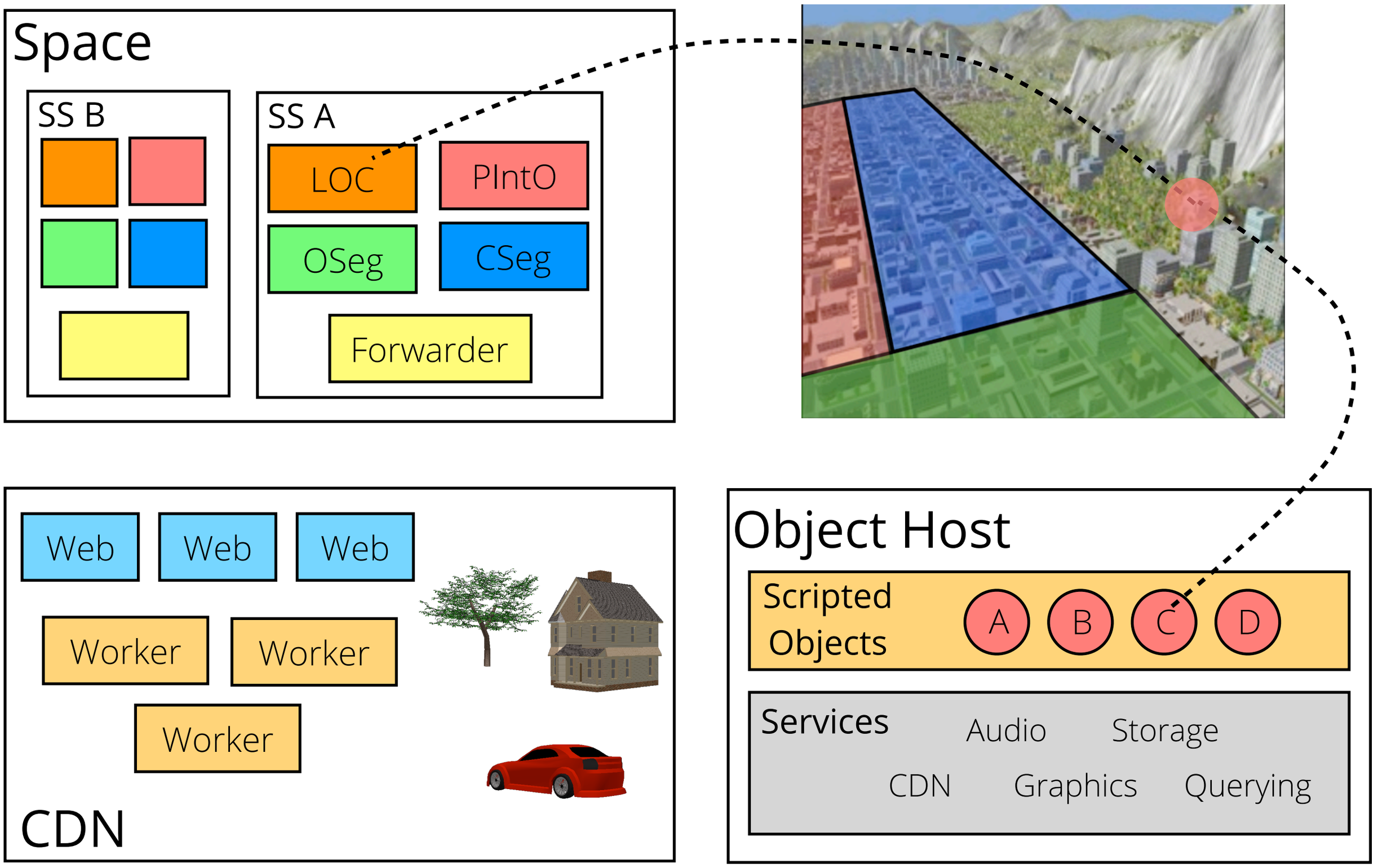




At a high level, the Sirikata architecture is straightforward and should look familiar if you've ever built a virtual world application. The object host, multiple of which can connect, provide basic object simulation by running object scripts. They connect to the space in order to interact with each other. The space tracks their basic properties like location and mesh, provides querying to let objects discover each other, and message forwarding for interaction. The space is broken down by another service, CSeg. Each region is managed by a space server. Finally, the CDN, or content distribution network, manages large static content like meshes or audio clips. The name is now a bit of a misnomer: besides distributing this data through HTTP frontend servers, it also accepts uploads, and optimizes them for progressive download and display. At a high level, you can think of these three components provide 3 abstractions: the object host provides computation, the space provides communication, and the CDN provides storage.

As a quick example of how these components interact, consider an object on an object host. When it's loaded it has no physical representation. If it wants to add itself to the world as a tree, it would initiate session with the appropriate space server and request the location and mesh. This would be added to the space servers *location service*. The location service would store a URL reference to the mesh on the CDN. If an avatar, which is just an object, views the world, it will query *PIntO* for *potentially interesting objects*. When it gets the tree object, it uses the URL to go to the CDN and download the mesh.

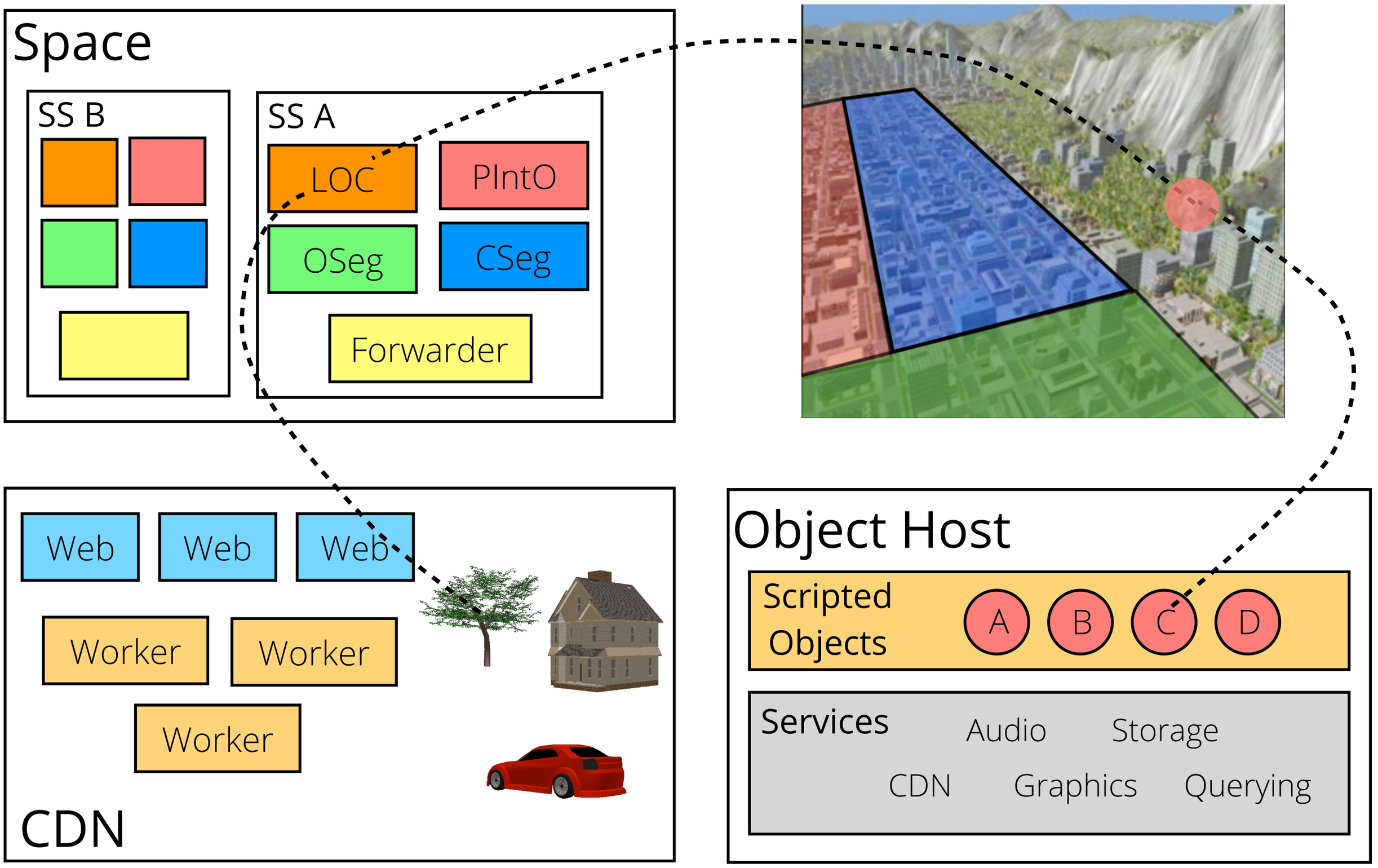
Sirikata Architecture



At a high level, the Sirikata architecture is straightforward and should look familiar if you've ever built a virtual world application. The object host, multiple of which can connect, provide basic object simulation by running object scripts. They connect to the space in order to interact with each other. The space tracks their basic properties like location and mesh, provides querying to let objects discover each other, and message forwarding for interaction. The space is broken down by another service, CSeg. Each region is managed by a space server. Finally, the CDN, or content distribution network, manages large static content like meshes or audio clips. The name is now a bit of a misnomer: besides distributing this data through HTTP frontend servers, it also accepts uploads, and optimizes them for progressive download and display. At a high level, you can think of these three components provide 3 abstractions: the object host provides computation, the space provides communication, and the CDN provides storage.

As a quick example of how these components interact, consider an object on an object host. When it's loaded it has no physical representation. If it wants to add itself to the world as a tree, it would initiate session with the appropriate space server and request the location and mesh. This would be added to the space servers *location service*. The location service would store a URL reference to the mesh on the CDN. If an avatar, which is just an object, views the world, it will query *PIntO* for *potentially interesting objects*. When it gets the tree object, it uses the URL to go to the CDN and download the mesh.

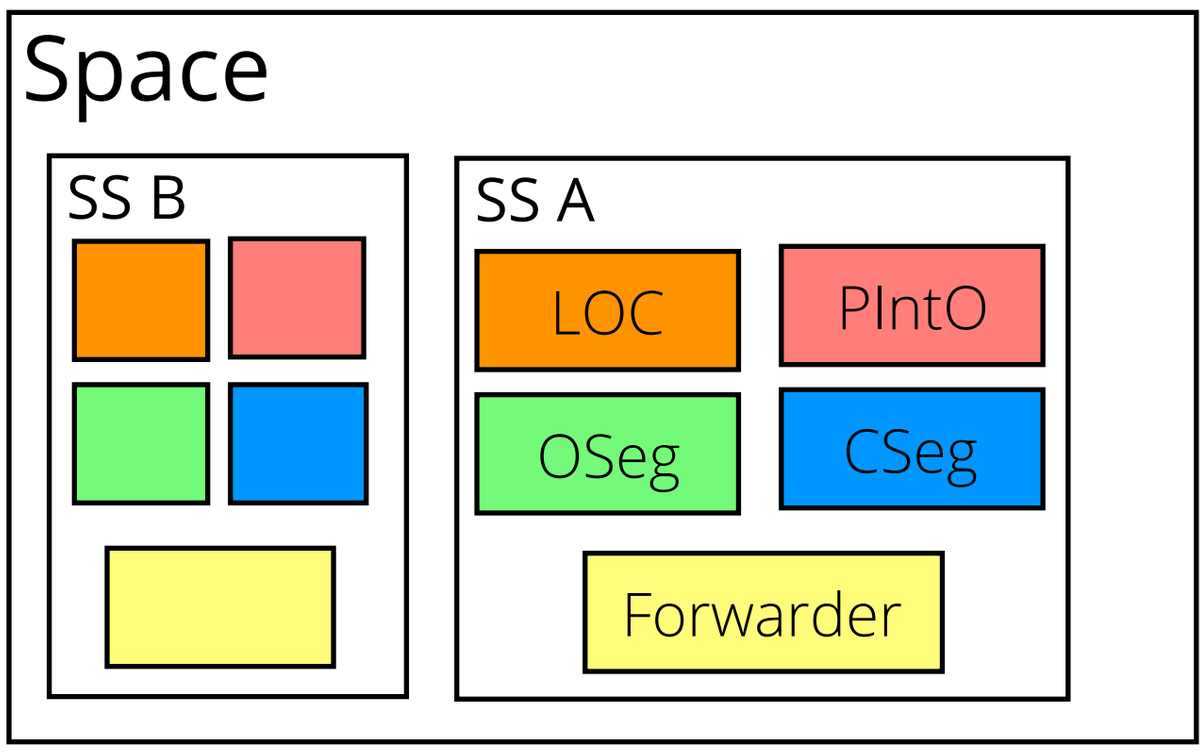
Sirikata Architecture

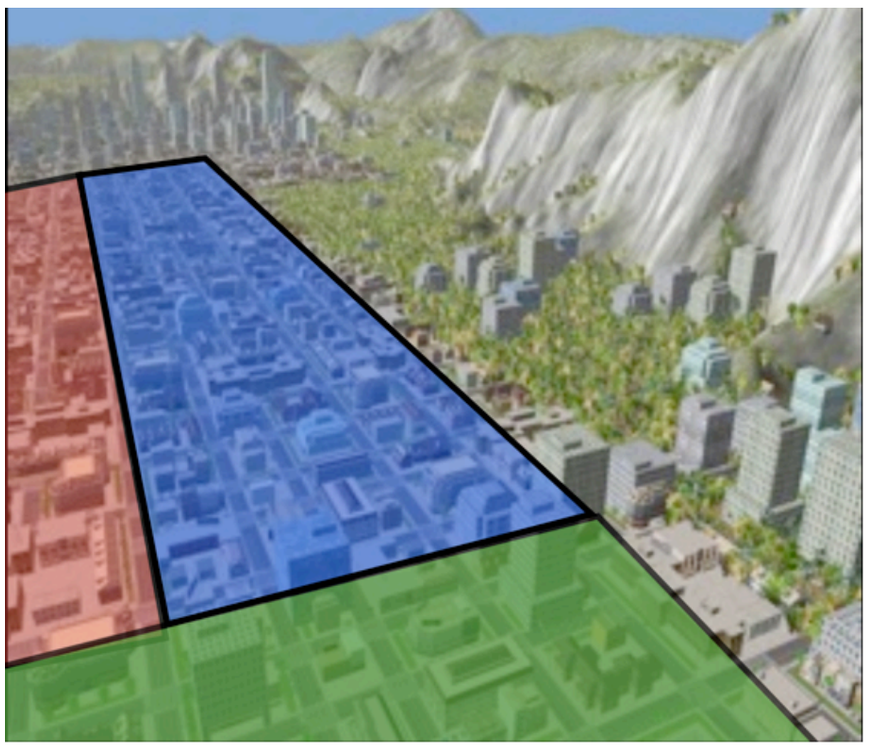


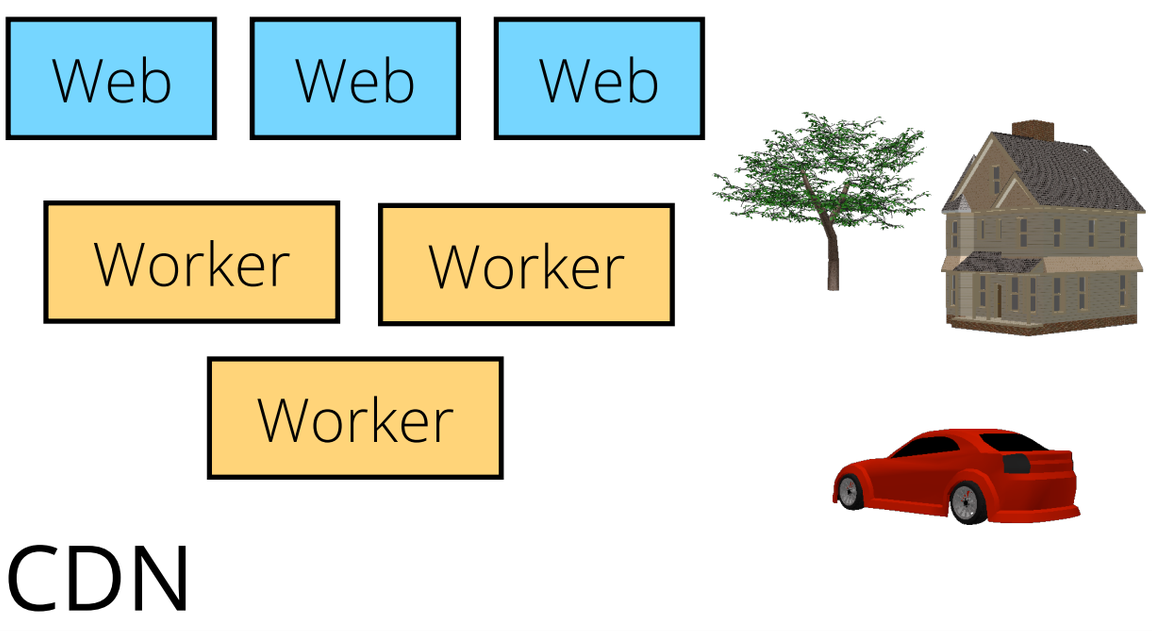
At a high level, the Sirikata architecture is straightforward and should look familiar if you've ever built a virtual world application. The object host, multiple of which can connect, provide basic object simulation by running object scripts. They connect to the space in order to interact with each other. The space tracks their basic properties like location and mesh, provides querying to let objects discover each other, and message forwarding for interaction. The space is broken down by another service, CSeg. Each region is managed by a space server. Finally, the CDN, or content distribution network, manages large static content like meshes or audio clips. The name is now a bit of a misnomer: besides distributing this data through HTTP frontend servers, it also accepts uploads, and optimizes them for progressive download and display. At a high level, you can think of these three components provide 3 abstractions: the object host provides computation, the space provides communication, and the CDN provides storage.

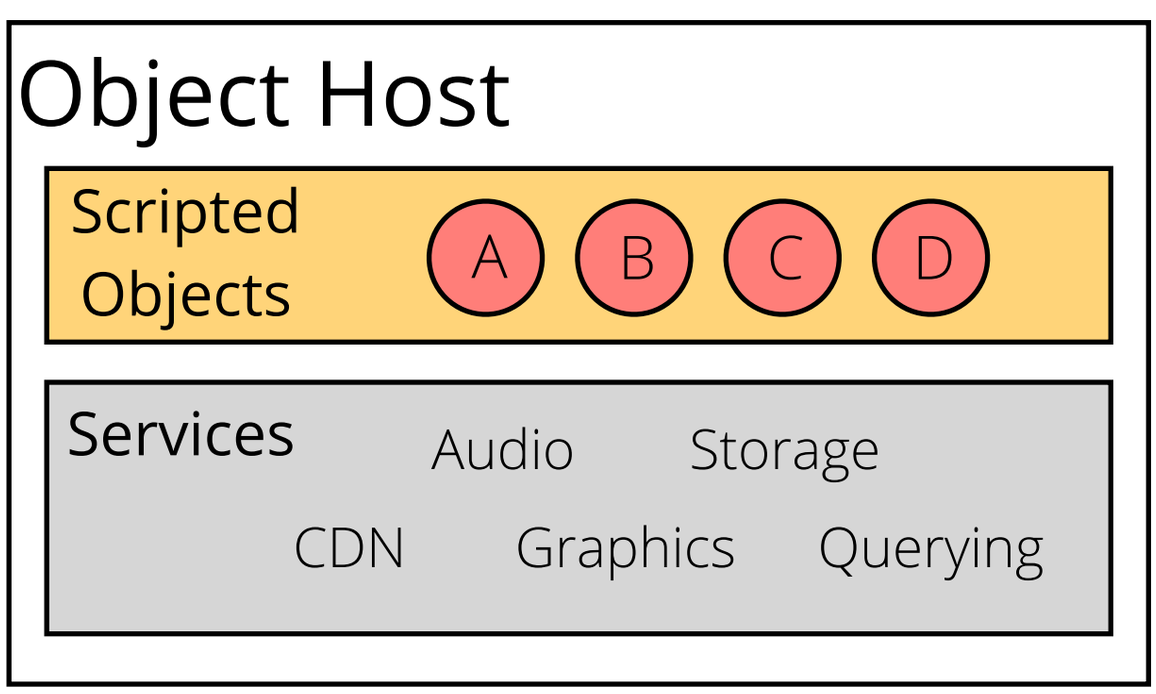
As a quick example of how these components interact, consider an object on an object host. When it's loaded it has no physical representation. If it wants to add itself to the world as a tree, it would initiate session with the appropriate space server and request the location and mesh. This would be added to the space servers *location service*. The location service would store a URL reference to the mesh on the CDN. If an avatar, which is just an object, views the world, it will query *PIntO* for *potentially interesting objects*. When it gets the tree object, it uses the URL to go to the CDN and download the mesh.

Sirikata Architecture





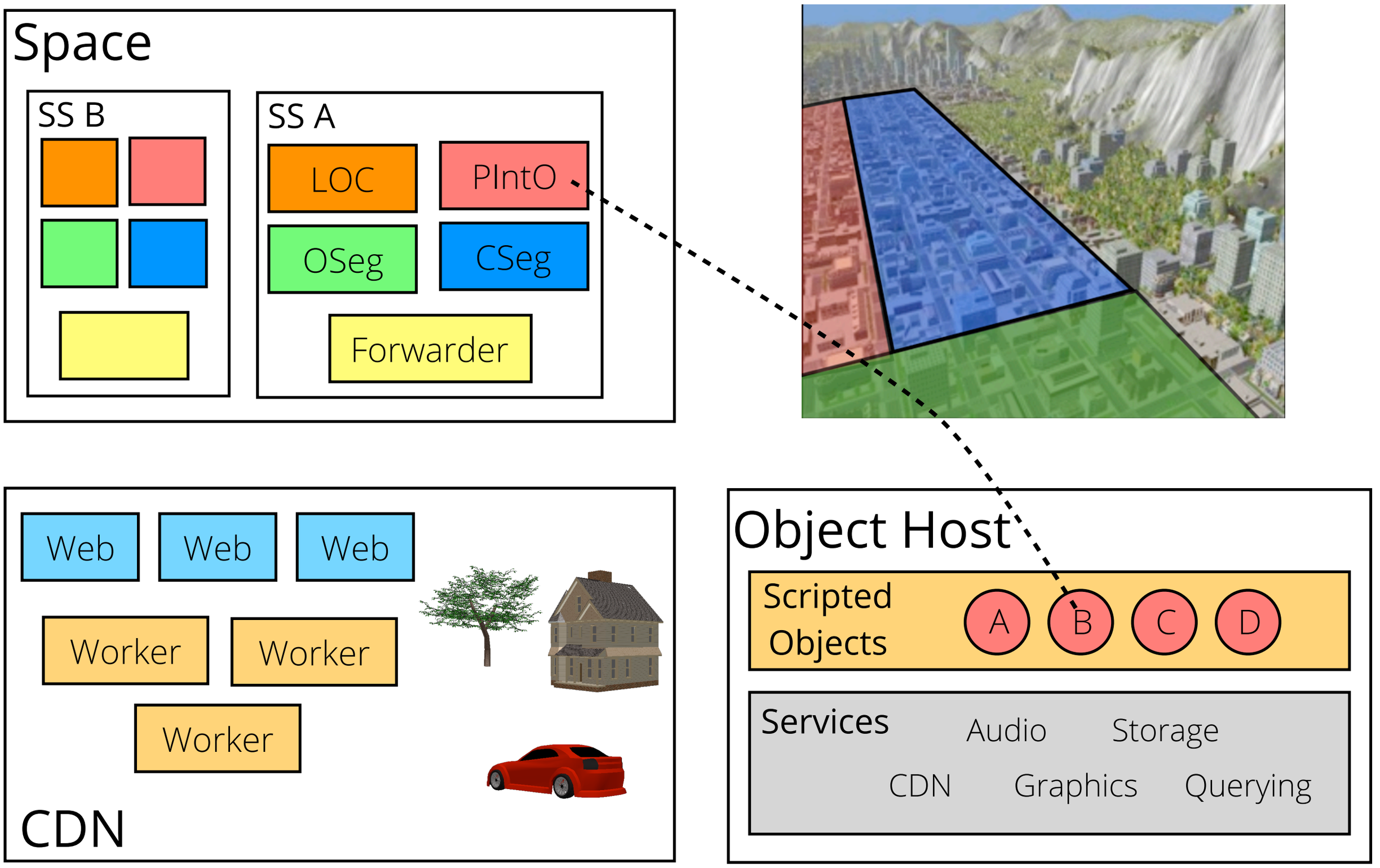




At a high level, the Sirikata architecture is straightforward and should look familiar if you've ever built a virtual world application. The object host, multiple of which can connect, provide basic object simulation by running object scripts. They connect to the space in order to interact with each other. The space tracks their basic properties like location and mesh, provides querying to let objects discover each other, and message forwarding for interaction. The space is broken down by another service, CSeg. Each region is managed by a space server. Finally, the CDN, or content distribution network, manages large static content like meshes or audio clips. The name is now a bit of a misnomer: besides distributing this data through HTTP frontend servers, it also accepts uploads, and optimizes them for progressive download and display. At a high level, you can think of these three components provide 3 abstractions: the object host provides computation, the space provides communication, and the CDN provides storage.

As a quick example of how these components interact, consider an object on an object host. When it's loaded it has no physical representation. If it wants to add itself to the world as a tree, it would initiate session with the appropriate space server and request the location and mesh. This would be added to the space servers *location service*. The location service would store a URL reference to the mesh on the CDN. If an avatar, which is just an object, views the world, it will query *PIntO* for *potentially interesting objects*. When it gets the tree object, it uses the URL to go to the CDN and download the mesh.

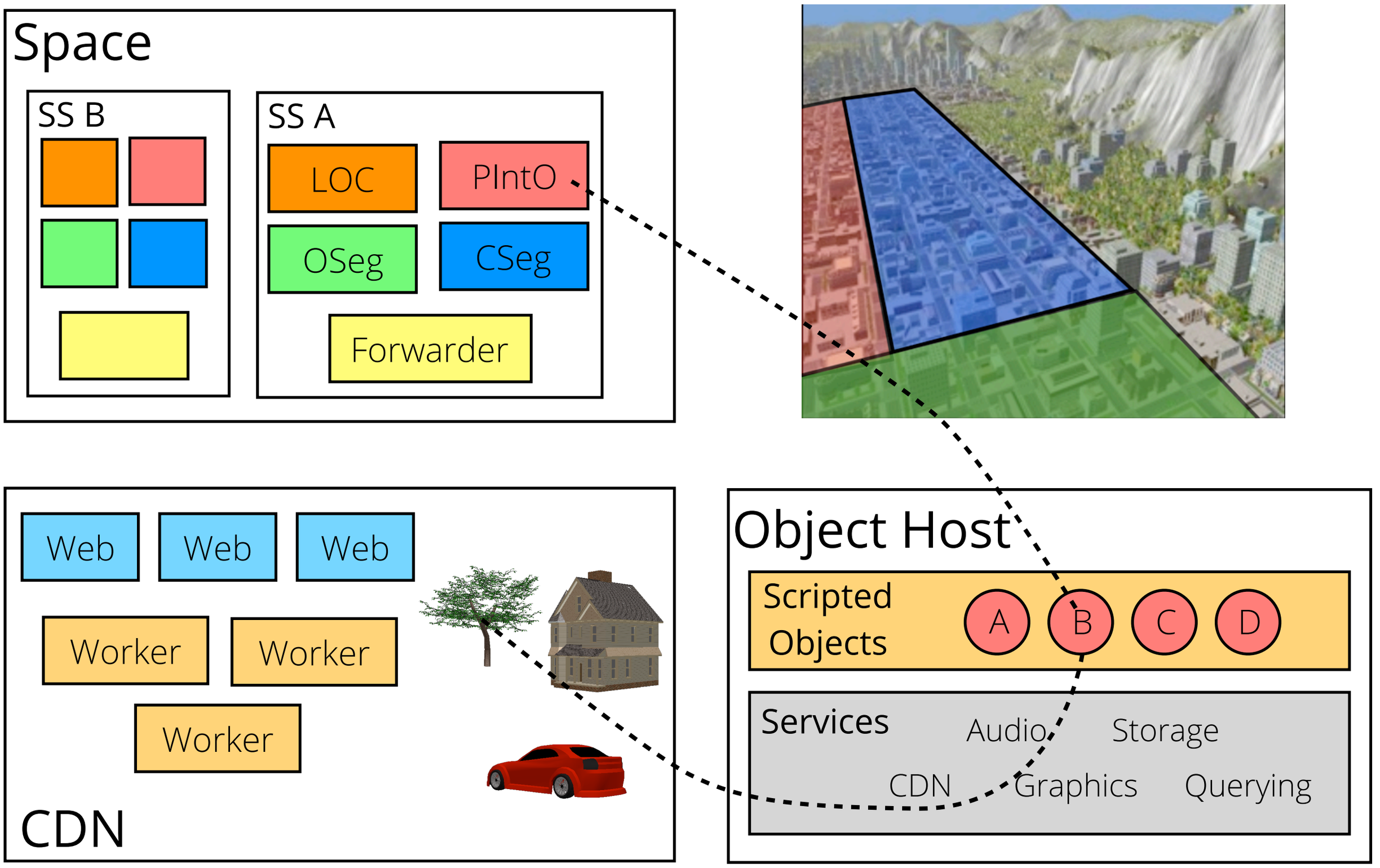
Sirikata Architecture



At a high level, the Sirikata architecture is straightforward and should look familiar if you've ever built a virtual world application. The object host, multiple of which can connect, provide basic object simulation by running object scripts. They connect to the space in order to interact with each other. The space tracks their basic properties like location and mesh, provides querying to let objects discover each other, and message forwarding for interaction. The space is broken down by another service, CSeg. Each region is managed by a space server. Finally, the CDN, or content distribution network, manages large static content like meshes or audio clips. The name is now a bit of a misnomer: besides distributing this data through HTTP frontend servers, it also accepts uploads, and optimizes them for progressive download and display. At a high level, you can think of these three components provide 3 abstractions: the object host provides computation, the space provides communication, and the CDN provides storage.

As a quick example of how these components interact, consider an object on an object host. When it's loaded it has no physical representation. If it wants to add itself to the world as a tree, it would initiate session with the appropriate space server and request the location and mesh. This would be added to the space servers *location service*. The location service would store a URL reference to the mesh on the CDN. If an avatar, which is just an object, views the world, it will query *PIntO* for *potentially interesting objects*. When it gets the tree object, it uses the URL to go to the CDN and download the mesh.

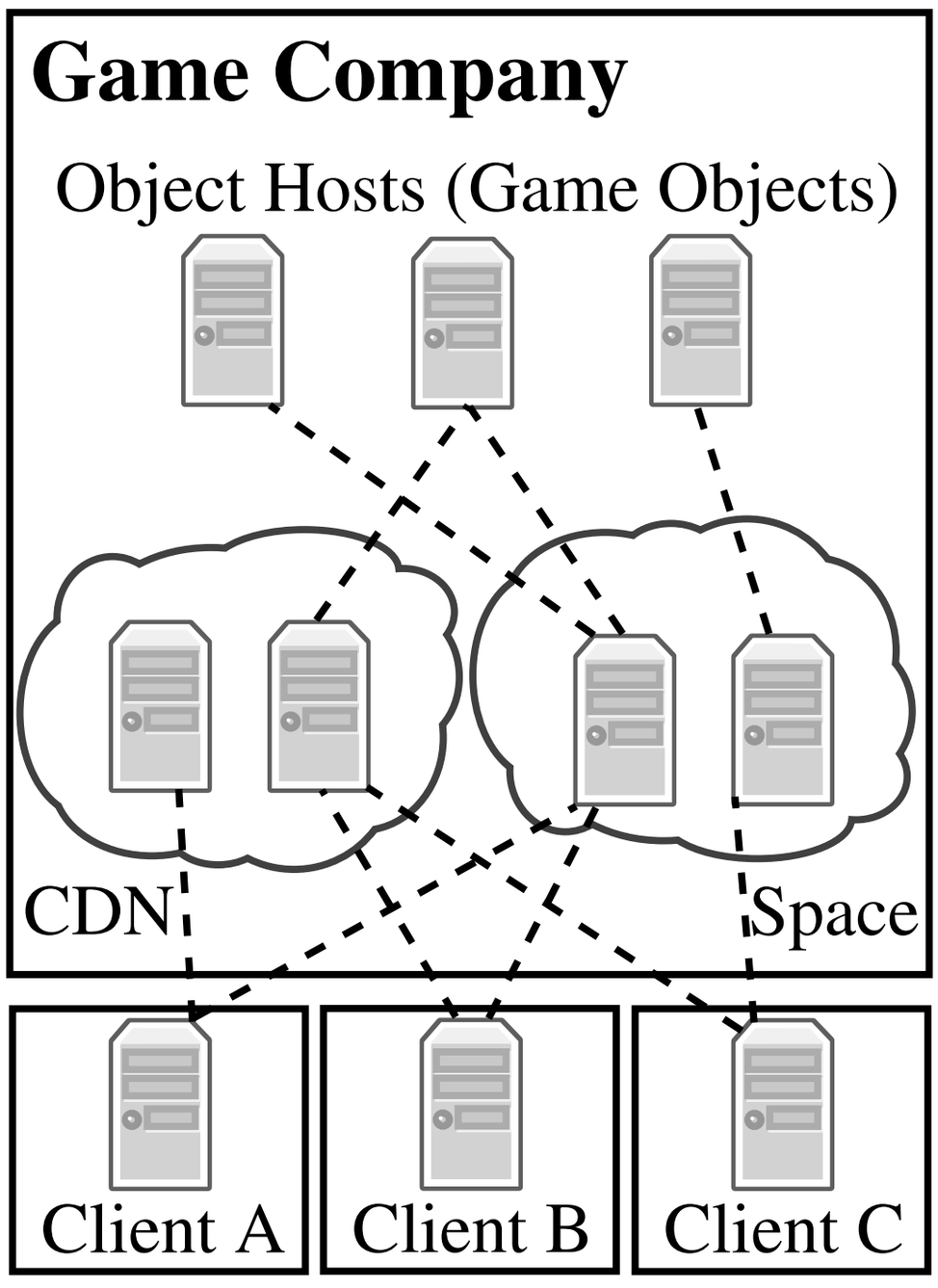
Sirikata Architecture

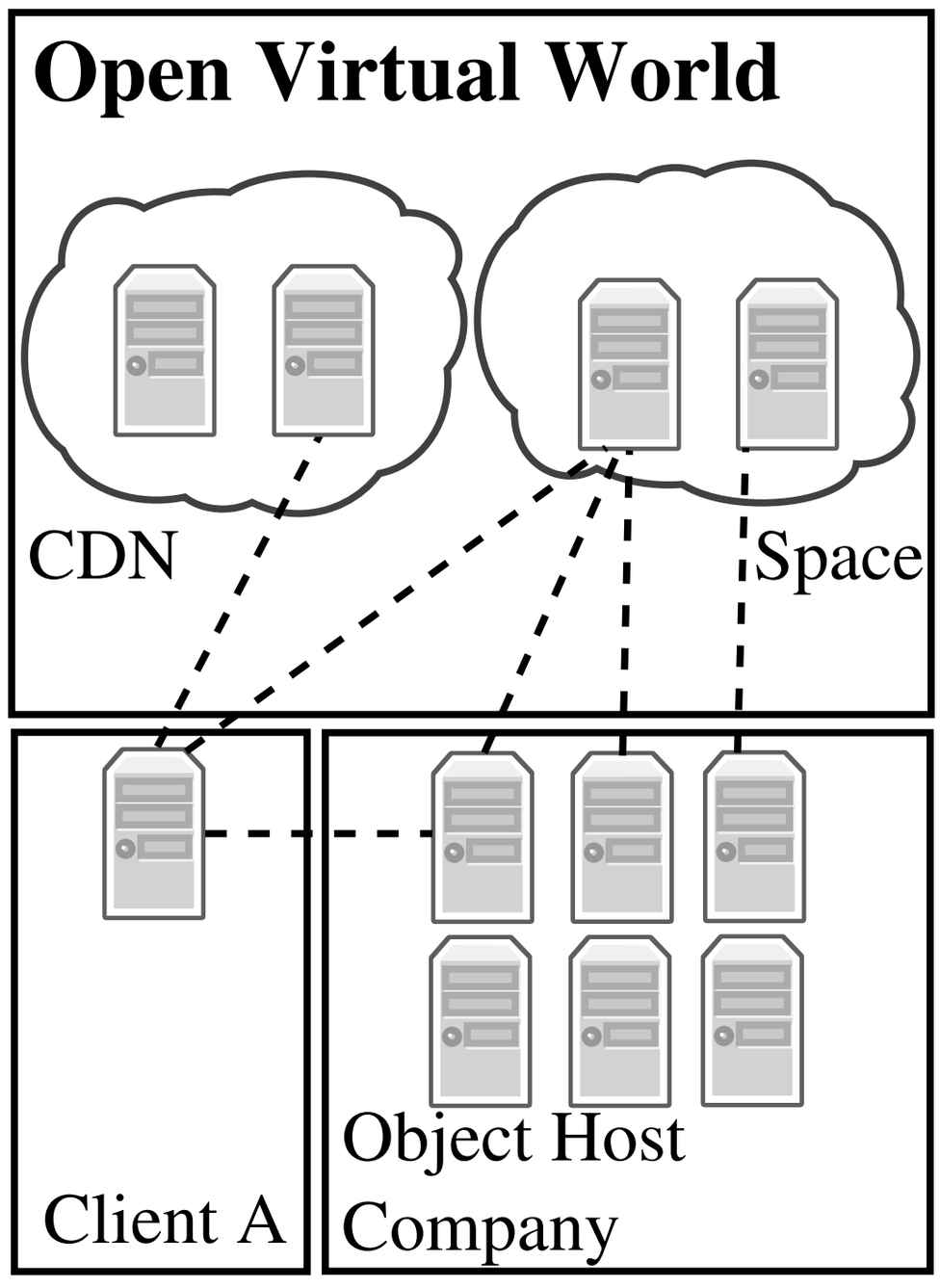


At a high level, the Sirikata architecture is straightforward and should look familiar if you've ever built a virtual world application. The object host, multiple of which can connect, provide basic object simulation by running object scripts. They connect to the space in order to interact with each other. The space tracks their basic properties like location and mesh, provides querying to let objects discover each other, and message forwarding for interaction. The space is broken down by another service, CSeg. Each region is managed by a space server. Finally, the CDN, or content distribution network, manages large static content like meshes or audio clips. The name is now a bit of a misnomer: besides distributing this data through HTTP frontend servers, it also accepts uploads, and optimizes them for progressive download and display. At a high level, you can think of these three components provide 3 abstractions: the object host provides computation, the space provides communication, and the CDN provides storage.

As a quick example of how these components interact, consider an object on an object host. When it's loaded it has no physical representation. If it wants to add itself to the world as a tree, it would initiate session with the appropriate space server and request the location and mesh. This would be added to the space servers *location service*. The location service would store a URL reference to the mesh on the CDN. If an avatar, which is just an object, views the world, it will query *PIntO* for *potentially interesting objects*. When it gets the tree object, it uses the URL to go to the CDN and download the mesh.

Configurations





Supports different virtual world applications

This breakdown actually appears in most of these systems, but Sirikata requires that each of these components is a separate networked service. Combining them in a single process may work for small scale games, but doesn't work for large, open, metaverse-style games, or is at least a hindrance to scalability.

One of the major motivations and benefits of this breakdown is that we can support different configurations for different applications. Here I've shown how we can support both a game application with centralized control of everything but the client and an open metaverse style world where users can host their own objects in a truly federated system.

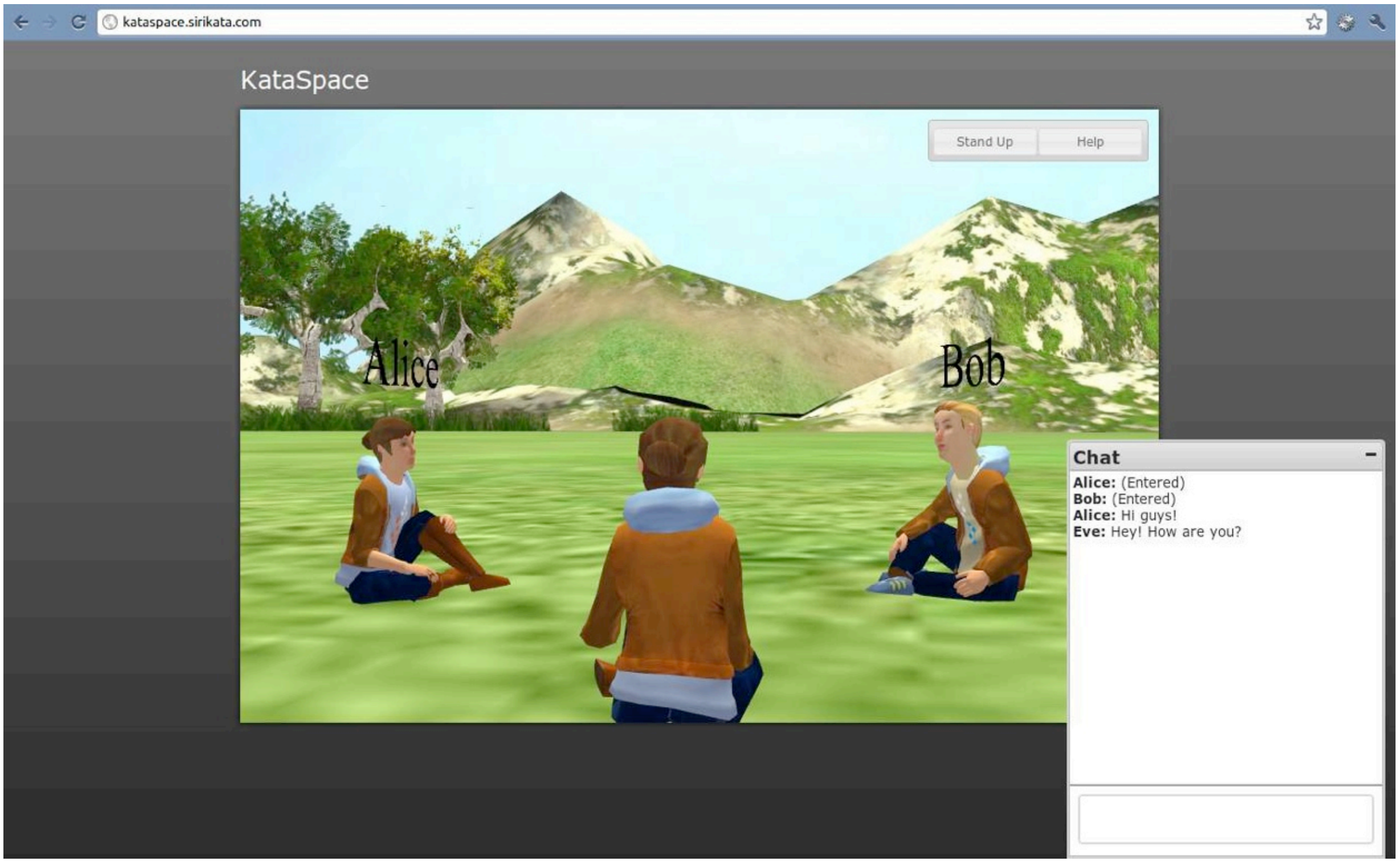
Porting to the Web

- Swapping implementations of services is easy
- Focus on web-based clients
- Goal: Browser-based object host

When we decided to try porting a client to the web browser, our task was made easier because of this system decomposition -- everything was already setup to make swapping out different implementations easy. Of course, until you've actually swapped out a component you're never sure just how coupled the existing implementations are.

Our initial focus in moving to the web was to make it possible to experience a Sirikata based world in the client. Since clients of Sirikata are just object hosts that render the world, our task was to build a browser-based object host.

KataSpace

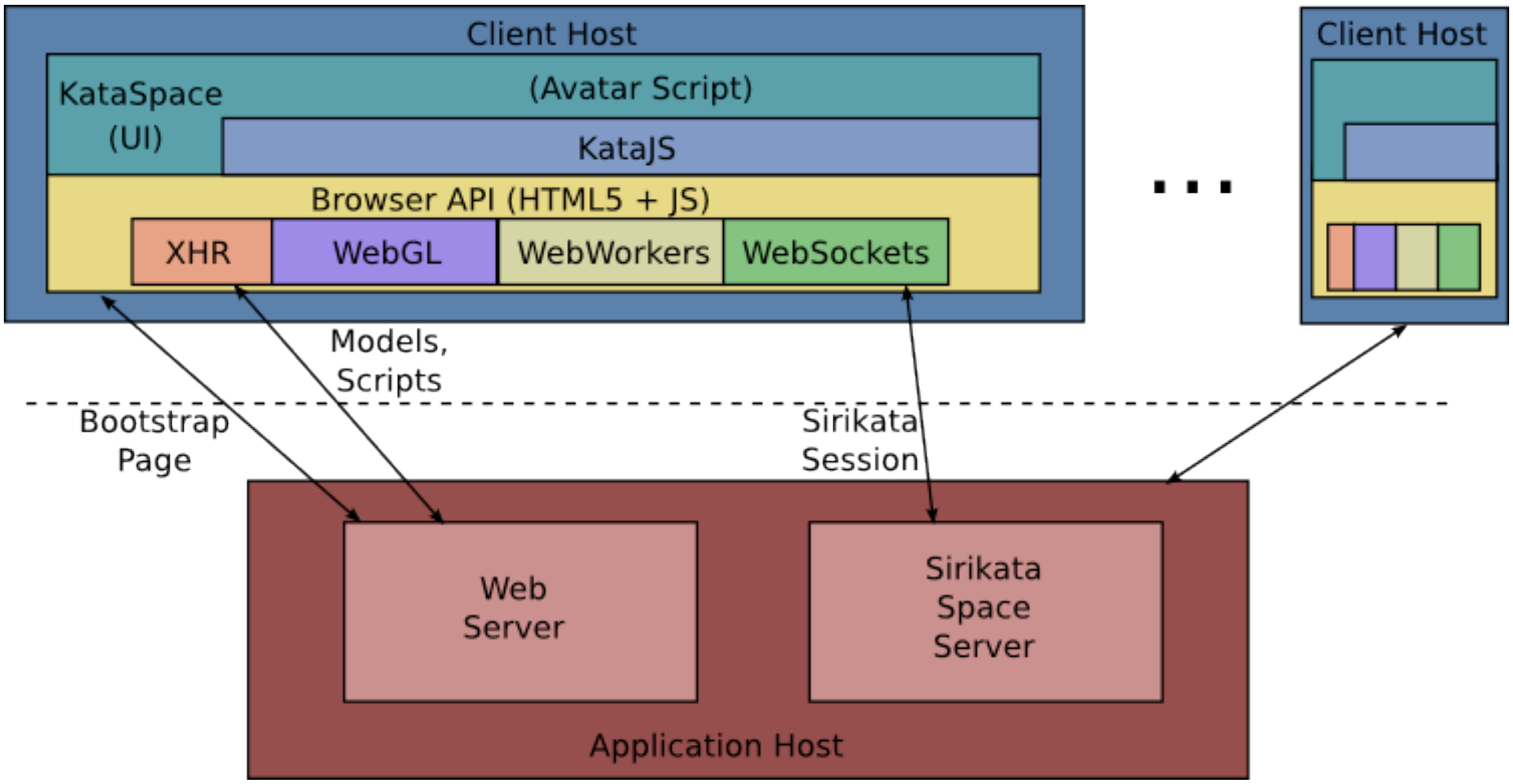


Thanks to Daniel Horn, Patrick Horn, & Daniel Miller for ProtoJS, KataJS, and KataSpace development

In late 2010 and early 2011 we did the majority of the work of building a Sirikata-compatible object host, including display and interaction. The demo application we built was called KataSpace and was intentionally simple -- just avatars sharing a space, walking around, and chatting. This is a pretty minimal application you can quickly build on top of the basic functionality Sirikata provides.

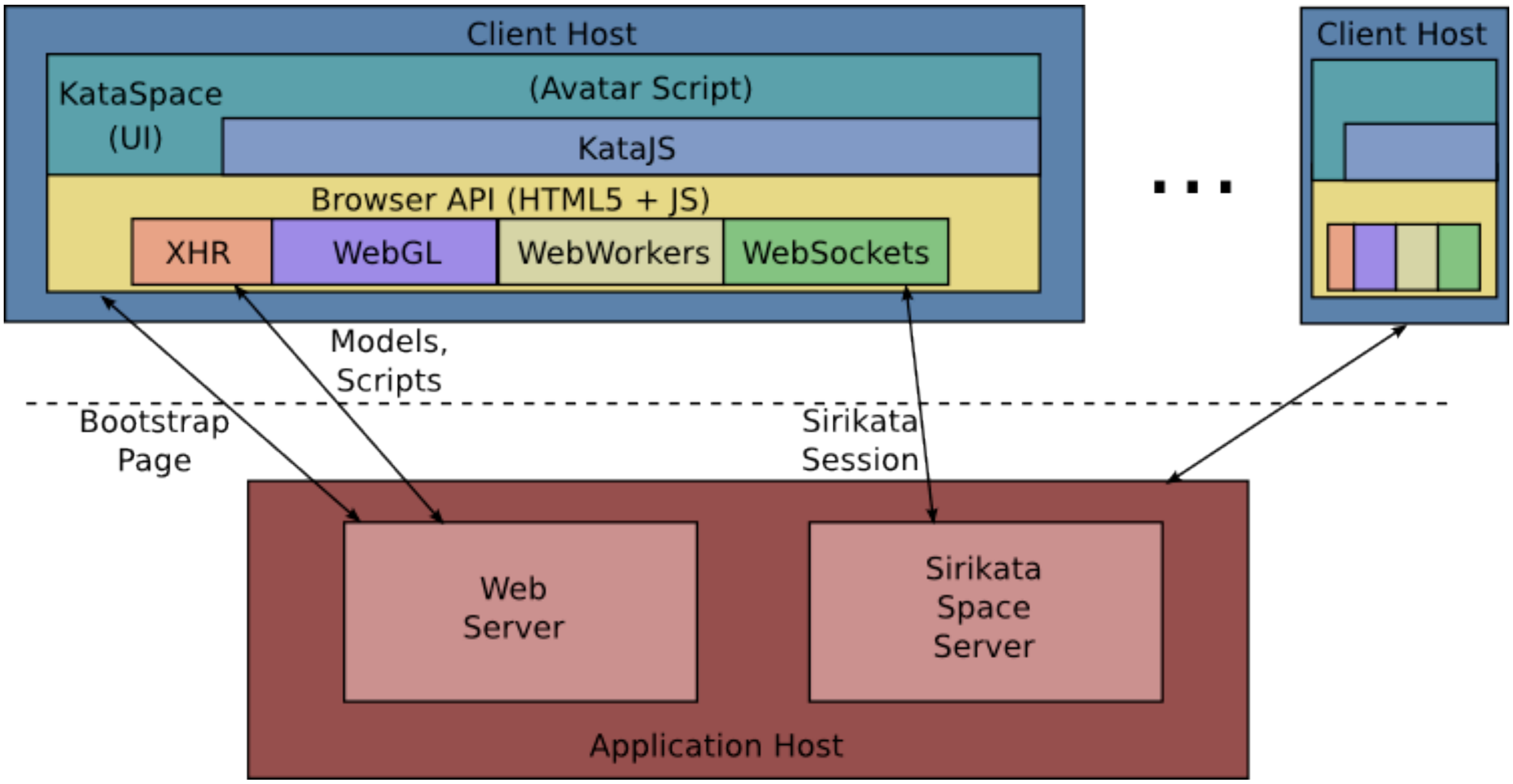
This was used as the basis of a real deployment for a project called the BE community, which aimed to give young patients with cancer an immersive, online, multi-user environment where they would have a chance to interact with others in similar conditions, alleviating the social isolation they frequently experience.

KataSpace Architecture



Here I've drawn an overview of the Kataspace architecture. At the bottom, you'll notice we're just using the stock Sirikata space server. This was enabled by adding WebSocket support to our transport layer. To the left, you can see we're using just a regular web server. This made building the initial prototype simpler, but the real CDN can be used as well.

KataSpace Architecture



The top of the figure shows the new stuff. We used many new technologies, including WebGL, WebWorkers, and WebSockets, and of course took advantage of existing APIs. Most of the code went into a reusable library called KataJS which implements an entire object host within the browser. It does session management, the interaction with basic space services like location management, querying for other objects, and messaging. It also has a few extensions, for example the renderer used to display the world, which “just work” by hooking into the core functionality.

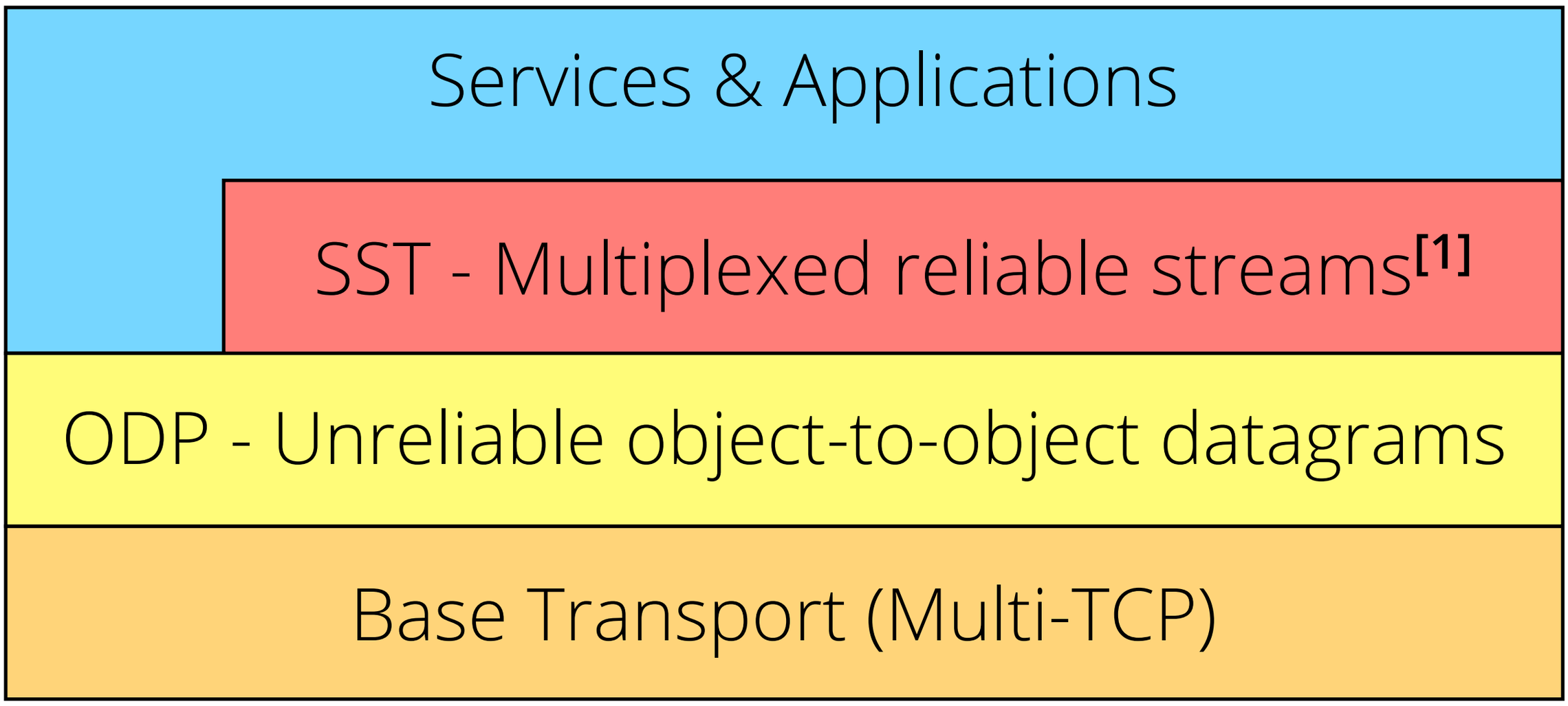
The actual application was pretty small, about 1500 lines of HTML and JavaScript which implemented the UI and the avatar script. The avatar script was pretty straightforward, essentially just dealing with input, the GUI, and chat messages.

Lessons Learned

- Performance characteristics
  - Custom protocols can be expensive
  - Mesh conditioning and aggregation

In a sense this port was easy -- the new APIs provided equivalents to everything we had in C++. The problem was not with functionality but performance. Unfortunately, both key tasks for a web based client were slow: dealing with the basic messages that needed to be sent and received and loading and rendering of meshes were both too slow. I'll talk about each of these in a bit more detail.

Sirikata Protocol Stack

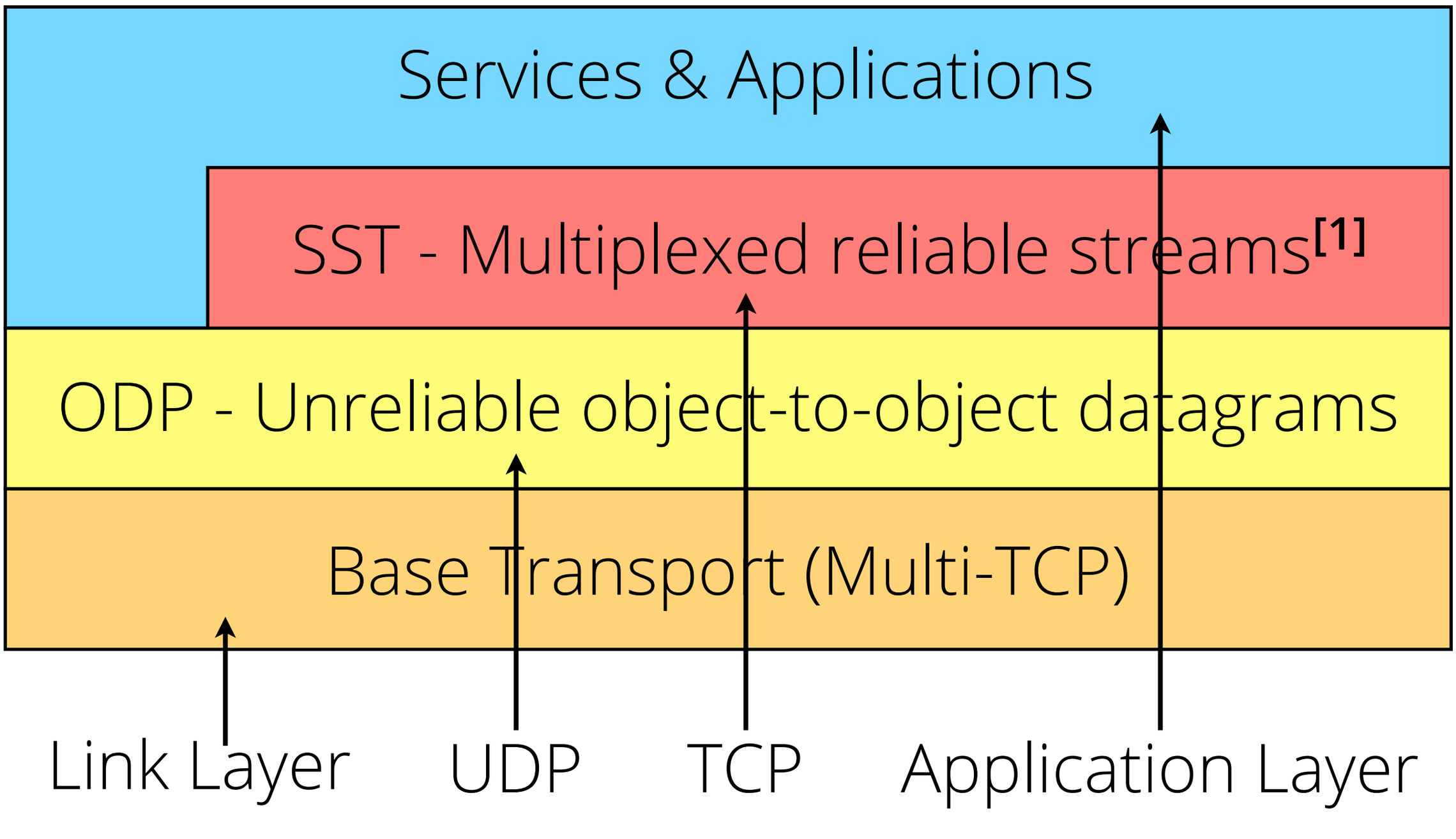


[1] Structured Streams: A New Transport Abstraction. Ford. SIGCOMM 2007

So let's start out with what Sirikata's protocol stack looks like. At the base we have the protocol used to connect two nodes. In theory this is pluggable, but we really only support a multiple-TCP-connection based approach that is reliable, can make connections anywhere, but avoids head-of-line-blocking. On top of that we have our core unreliable object-to-object datagram protocol, ODP. For reliable streams we use an implementation of structured streams, which gives you reliable streams like TCP but shares an underlying congestion control layer, makes creating new streams very cheap, and actually gives you even more, such as arbitrarily sized reliable and unreliable datagrams. Finally, services and applications build on either ODP or SST. For example, the basic session protocol management and the chat protocol in KataSpace work over ODP and SST.

This particular breakdown bears a striking resemblance to another stack you're probably familiar with. This isn't a coincidence. The basic layers fall out similarly because at its core, Sirikata just forms an overlay network where objects are the endpoints.

Sirikata Protocol Stack

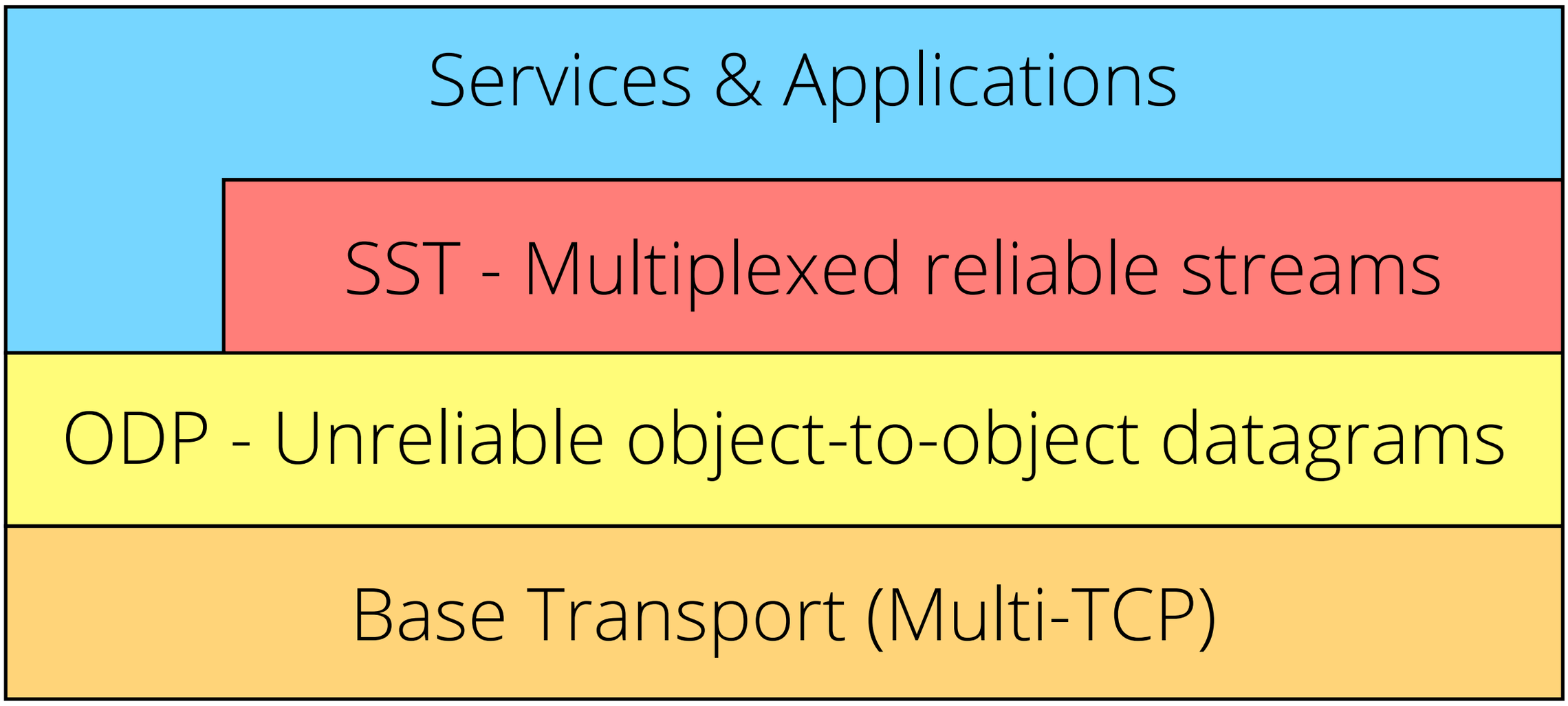


[1] Structured Streams: A New Transport Abstraction. Ford. SIGCOMM 2007

So let's start out with what Sirikata's protocol stack looks like. At the base we have the protocol used to connect two nodes. In theory this is pluggable, but we really only support a multiple-TCP-connection based approach that is reliable, can make connections anywhere, but avoids head-of-line-blocking. On top of that we have our core unreliable object-to-object datagram protocol, ODP. For reliable streams we use an implementation of structured streams, which gives you reliable streams like TCP but shares an underlying congestion control layer, makes creating new streams very cheap, and actually gives you even more, such as arbitrarily sized reliable and unreliable datagrams. Finally, services and applications build on either ODP or SST. For example, the basic session protocol management and the chat protocol in KataSpace work over ODP and SST.

This particular breakdown bears a striking resemblance to another stack you're probably familiar with. This isn't a coincidence. The basic layers fall out similarly because at its core, Sirikata just forms an overlay network where objects are the endpoints.

Porting Challenges



Protocols had design decisions incompatible with performant browser implementation

When we ported these layers to JavaScript to run in the browser, we ran into performance problems because their design made sense for native implementations but are difficult to implement efficiently in JavaScript, mainly because they require a lot of handling of binary data and bit manipulations. Both ODP and SST implementations use Google's Protocol buffers and we had to implement a JavaScript library to decode them. So really it wasn't the protocols themselves, it was their encoding that was the issue.

Unfortunately, we also aren't willing to give up these encodings because the native services perform a lot better with them and that's worth when trying to scale to a large world -- it means fewer servers are required to run a world of the same size.

Protocols For All Platforms

- Multi-encoding support
  - JSON encoding?
  - Requires negotiation, but once per object host
  - Already abstract encoding, but statically

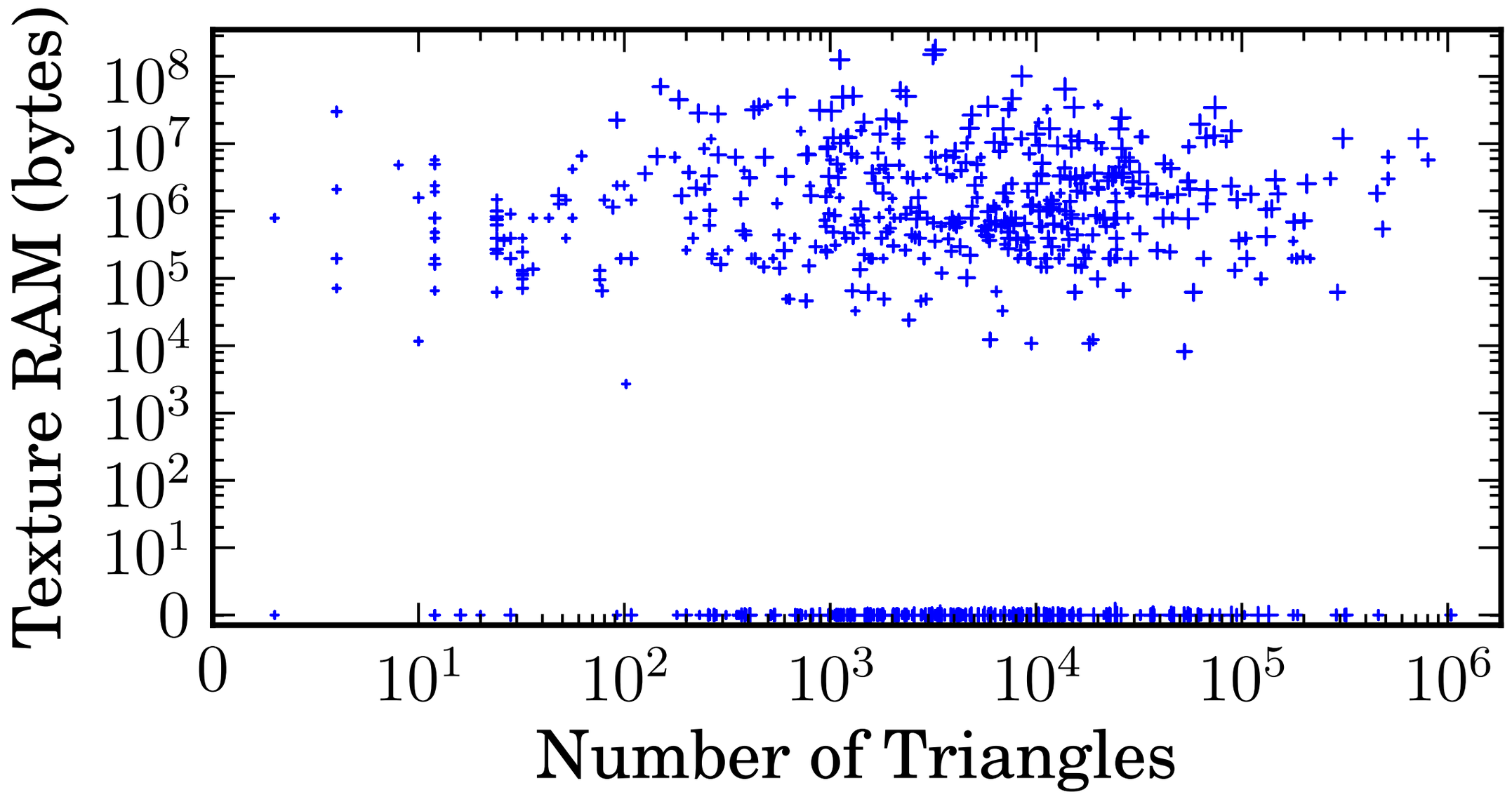
Long term, it seems to make sense to support multiple encodings for our protocols, delivering something like JSON to the browser. There are two issues to address. First, we need to select which encoding to use, but this probably only requires a once-per-connection negotiation. Second, the system needs to support both encodings. It turns out we already actually abstract away the encoding used for protocol messages, but only do that at compile time. We'd need to update this to work dynamically.

Lessons Learned

- Performance characteristics
  - Custom protocols can be expensive
  - Mesh conditioning and aggregation

So we might have a solution to expensive protocols as long as getting native decoding provides a sufficient improvement despite possibly larger message size. The other performance issue we ran into was with graphics. There are two aspects to this: the conditioning and format of the meshes delivered to the client, and aggregation. First let me explain what we do for our native C++ client.

Content Conditioning

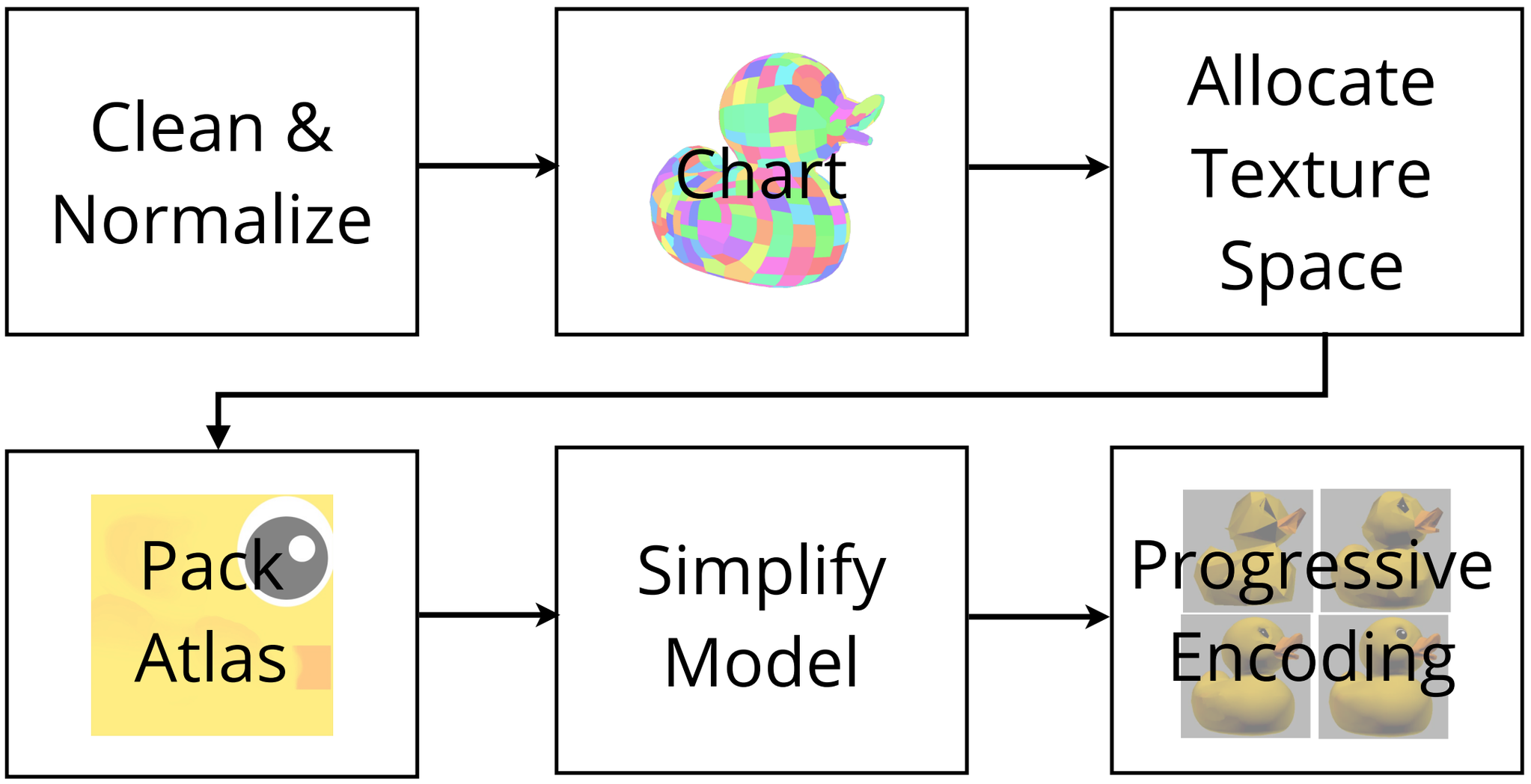


User models have huge range of properties.
(open3dhub.com)

Whether you're just trying to get assets created by an artist into your application or you support user generated content like a metaverse does, you need to do something to handle the wide variety of content you'll get. This graph shows the variation for just two metrics for a set of models collected from open3dhub.com, our repository for 3D models to be used in Sirikata worlds. Note the log axes. Of course quality levels, formats, exporter-specific customizations and features will also vary widely.

Content Conditioning

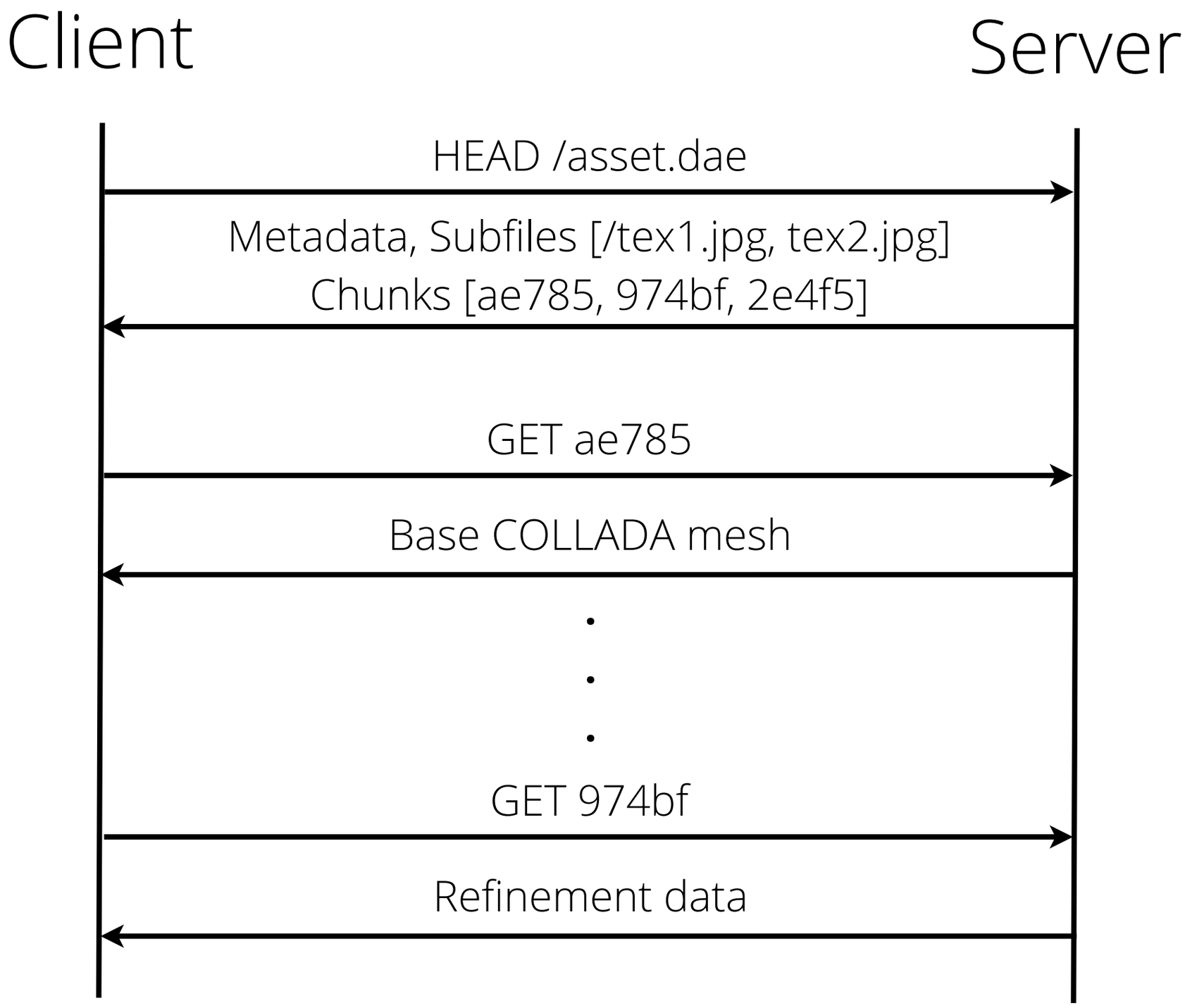
Sirikata's Unsupervised Conditioning Pipeline [1]



[1] Unsupervised Conversion of 3D Models for Interactive Metaverses. Terrace et al. ICME 2012

Sirikata uses a content conditioning pipeline on it's CDN, where all content is uploaded before it is used in the world. The pipeline is unsupervised and produces models appropriate for rendering by clients. Sirikata's pipeline is especially tricky because we take **any** model from users, so it needs to be very robust. You should see the paper for details, but the output is a possibly simplified mesh with atlased and compressed textures and a progressive encoding split across multiple independently cacheable chunks.

Content Delivery



Once we've got the content in a form that we're confident clients can handle, we need to deliver it. Here we take advantage of existing tools. Content is replicated, you look up a nearby copy with DNS, and then start pulling content over HTTP.

But it's not just HTTP. For caching and streaming purposes we separate an initial metadata request, shown at the top, from the data. The data is referenced and cached by hash, which provides deduplication across meshes which is actually very important because we see a lot of reuse. The progressive mesh is split into a base mesh followed by chunks of refinement data, which may or may not be downloaded depending on what the client can support displaying.

Content on the Web

- Formats: prepared for GPU
  - Or at least native parsing [1,2,3]
  - Progressive?
  - To get interoperability and performance, we're going to need standardization

[1] Badgerfish. http://badgerfish.ning.com/
[2] webgl-loader. http://code.google.com/p/webgl-loader/
[3] blender-webgl-exporter. http://code.google.com/p/blender-webgl-exporter/

So how does all this map to the web, specifically if you're pulling content into a web-based client?

The loading process was one of the biggest problems for KataJS and KataSpace, leading to long parsing delays. We tried to use COLLADA which is great as an interchange format, but not so much as a delivery format. It's just too expensive for clients to parse, especially since many values need to be parsed from their human readable form. Even just Badgerfish, a simple JSON encoding of COLLADA would probably have been a huge improvement.

More recently we've seen this changing as there has been a small explosion in delivery formats, usually loadable nearly directly onto the GPU and usually specific to the particular renderer being used. Unfortunately this is quickly heading towards the challenges we have with the explosion of regular 3D file formats.

We also really need to address the progressive mesh issue. This is a necessity on the web since everything is loaded dynamically. Previously, only more niche applications had this requirement. It was encouraging to see a few papers at this conference about progressive loading and encoding of meshes, but it needs more attention and focus on standardization if we don't want to end up with this same explosion. The unique constraints of the web narrow the reasonable design choices, so pinning down a "web-based COLLADA" or "web-based delivery format" early on could be a huge boon for web-based 3D applications.

Content on the Web

- Already HTTP
  - Non-HTTP chunking scheme
  - Translate to ETag + range requests

In terms of transfer, it’s already HTTP, so that’s a good start. Unfortunately the custom chunking scheme doesn’t fit in as well, as it requires a custom approach to partial downloads. An alternative is to use a single file, range requests for chunking with ETags for caching, and still use a HEAD request to obtain metadata. The real challenge here is to figure out exactly how to map our scheme into HTTP to maximize caching. Unfortunately, there isn’t much on the web right now that matches this use case, so knowledge about the “right way” to do this for dynamically downloaded progressive content isn’t wide spread. Figuring this out and standardizing the approach would be beneficial to the entire community.

3D REST API

"REST 3D specifies an API to a web service that provides transactions and access to 3D assets in a manner independent of the underlying data storage.

It defines what a 3D asset is, and specify discovery, query, locking, and transaction operations."

- rest3d.org

SIGGRAPH BOF: August 7th 10 AM Rm 516

But what can we do beyond just the presentation and delivery of content? One of the interesting efforts in this domain is the 3D REST API. It's addresses all aspects of a content repository like our CDN: accessing, querying, and modifying content through a RESTful API. It's still early in development and really needs input from people with experience with systems like these. Sirikata's own site, open3dhub.com, is providing content in a web-friendly form, and has CORS enabled so you can integrate it into your own site, and we're hoping to implement the full 3D REST API as it's developed. If you're interested in this, there's a web site, mailing list, and, if you're attending SIGGRAPH, there's a Birds of a Feather session on Tuesday morning.

Aggregation

I want to also briefly mention aggregation. This moves beyond just the processing, presentation and download of individual meshes. Unlike games, our application can’t take a bunch of static content and pre-optimize it because our scenes are dynamic and user generated. Sirikata dynamically aggregates meshes automatically to enable the number of objects in the world to scale. Without aggregation, each object is reported individually and each mesh must be issued to the GPU separately unless the client can figure out some meshes to combine. Sirikata uses solid angle queries and aggregation to display more of the world with about the same cost. On screen is what you get from most systems dealing with very large worlds…

Aggregation

...and this is what you get from Sirikata. The entire world is displayed, though some at lower quality.

Aggregation

- Even more critical for web clients
  - Can’t aggregate locally and rendering cost per mesh is higher
- Decouple display and communication
  - Report mesh + list of object IDs
  - Still may report aggregate object IDs

Aggregation is even more critical for web clients because of performance limitations -- having the client find meshes that can be combined, merge meshes, and then push them to the GPU probably isn’t reasonable. One possible direction which we haven’t yet pursued is to decouple the reporting of objects for display and for communication. The querying service serves both purposes -- it returns object identifiers which can be used to communicate with and interact with other objects, and the properties of those objects, including position and mesh. If we at least partially decoupled these uses, essentially tracking 2 sets of results separately for display and communication, we might be able to display larger worlds without compromising the ability to interact with objects.

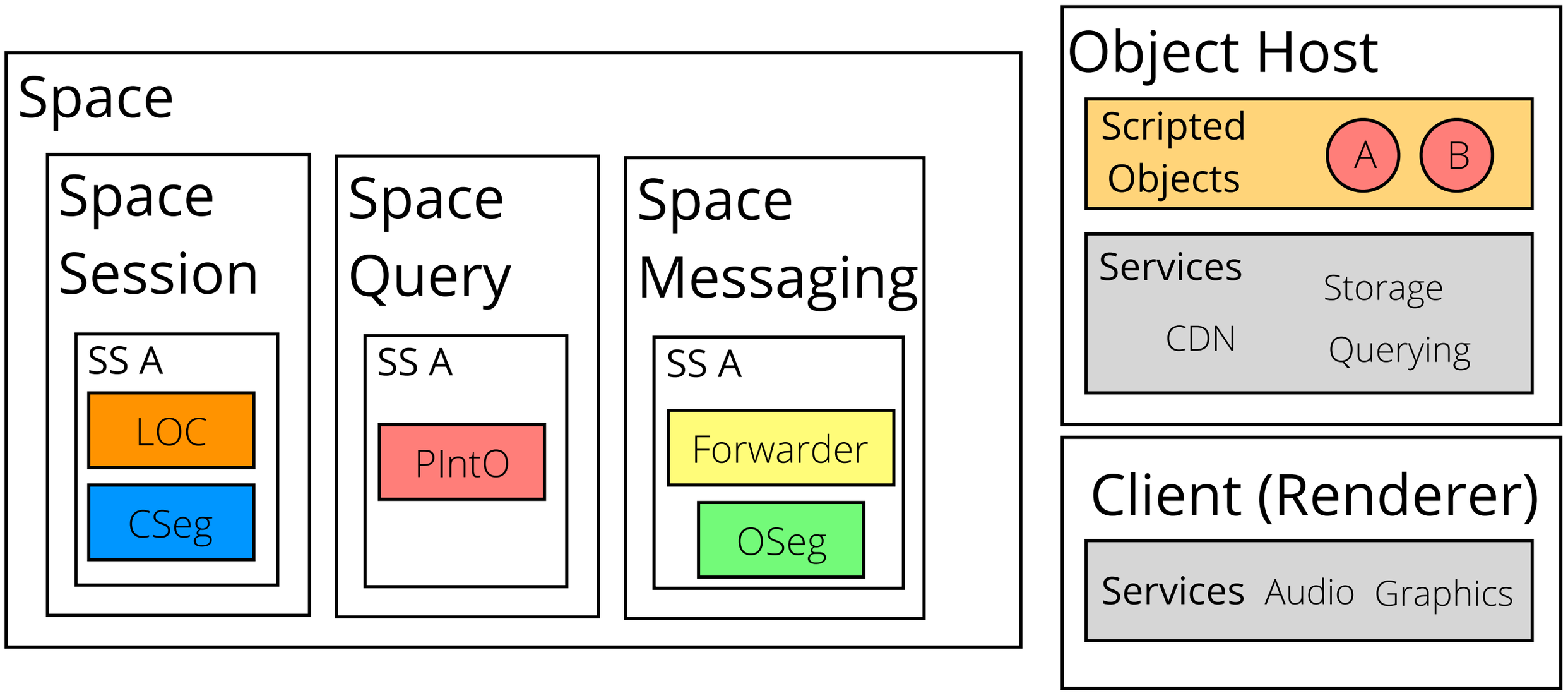
Evolving the Architecture

- Formalize multi-encoding support
  - Alternative pure HTTP base transport
- Content distribution network
  - Content processing as a service

So based on these challenges and the hints about fixes I've mentioned, where can we go? I'll quickly walk through a couple of directions I'm thinking about taking the Sirikata architecture. First, I mentioned multiple encoding formats. We need to formalize these and create the negotiation protocol. Besides the encoding I already talked about, we may also want to provide an alternative base transport based purely on HTTP, which would make building on existing HTTP stacks even easier.

I also mentioned modifying the content distribution network. The most important changes are to provide downloads and meshes which are more friendly to browsers, and both can be done as additions to the existing code. In fact, some of that is already there -- you can use models directly from the CDN using a simple download approach in KataSpace. More fundamentally, the CDN is moving more towards a content processing engine which could be interacted with as a web service. In fact, we already do this to upload aggregates and could certainly add more functionality like simplification.

Evolving the Architecture



Finer granularity services

Looking more at the high level architecture, we found that we would benefit from breaking down at least the object host a bit more because the web-based client is doing more than it absolutely needs to, which is a drawback performance-wise. Splitting object simulation and display so the web client is closer to a dumb terminal could be a substantial improvement.

More generally, we want to break down all of the services to finer granularity and possibly using a Cable Beach-like approach to pull them together. On the space server side, this could lead to a different breakdown where each logical service really is separate. That opens the possibility of interesting modifications like federated spaces with multiple implementations of services, for example messaging optimized for low-latency real time data versus more latency robust, larger bulk transfers, or different querying mechanisms, one for nearby objects and another more like a Google keyword search.

Evolving the Architecture

- Service oriented architecture
  - Everything is a service
  - RESTful querying, modification, messaging of the world
- Web-based virtual worlds mashups!

Finally, as the architecture becomes more strongly service-oriented architecture, we can expose more as web services. That's when I think we really become integrated with the web. What might users come up with when we can use OAuth access to our components, letting them compose them with other web services, and build new services on them? In terms of implementation, this is already happening because we've added HTTP control interfaces for our services.

Conclusion

- More to virtual worlds in the web than rendering
- Performance characteristics demand a shift in architecture
  - Supporting web and native clients creates tension
- True integration with the web means web-based interface, client, & linking

I want to conclude by making the point that what we found in building a web-based Sirikata client is that there's a lot more to building virtual worlds in the web than just writing the renderer. The web influences the rest of the system architecture. We saw this in how the performance of browsers requires different supporting services, and supporting both models creates a tension. But I think the most interesting work moving forward is looking at how to truly integrate with the web. We need to be looking at all aspects of this -- not just the standard client embedded in the browser, but all web-based or RESTful HTTP interfaces to worlds, DOM integration, and linking into and out of worlds and the rest of the web.

Thanks

More at sirikata.com

With that I'd like to say thanks for listening and, if you're interested in finding out more about Sirikata, check out our website. We think it's especially useful as a research platform since it's been designed with that in mind and just about everything is pluggable. If anyone has any questions, or suggestions, I'd be happy to take them now.