The Web as a Distributed Computing Platform

Nikos Vasilakis Pranjal Goel Henri Maxime Demoulin Jonathan M. Smith
The University of Pennsylvania

ABSTRACT

Perceived as a vast, interconnected graph of content, the reality of the web is very different. Immense computational resources are used to deliver this content and associated services. An even larger pool of computing power is comprised by edge user devices. This latent potential has gone unused. Ar frames the web as a distributed computing platform, unifying processing and storage infrastructure with a core programming model and a common set of browser-provided services. By exposing the inherent capacities to programmers, a far more powerful capability has been unleashed, that of the Internet as a distributed computing system. We have implemented a prototype system that, while modest in scale, fully illustrates what can be realized.

CCS CONCEPTS

• **Networks** → *World Wide Web (network structure)*; • **Computer systems organization** → *Peer-to-peer architectures*; • **Software and its engineering** → *Ultra-large-scale systems*; *Distributed systems organizing principles*;

KEYWORDS

Distribution, Internet, Web, JavaScript

1 INTRODUCTION

> *"The Internet as a computer... how do you write programs that compute over the Internet?"*
>
> Barbara Liskov, ACM Turing Award Lecture, 2008

The heterogeneity of user devices and the need for applications (*e.g.*, video or sophisticated graphics) have forced web browsers to assume a role resembling an operating system, with support for concurrency, resource management, and protected application multiplexing. Web browsers today feature several layers of high-performance machinery such as just-in-time compilers and state-of-the-art garbage collectors. Web applications have access to the same features as native applications do, including parallelism, ample memory and storage, and specialized hardware acceleration. JavaScript, the programming language underpinning web applications, stands alone in both its ubiquity and portability across browsers and platforms. These characteristics make the web an ideal basis for a distributed computing platform.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
EdgeSys'18, June 10–15, 2018, Munich, Germany
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5837-8/18/06...$15.00
https://doi.org/10.1145/3213344.3213346

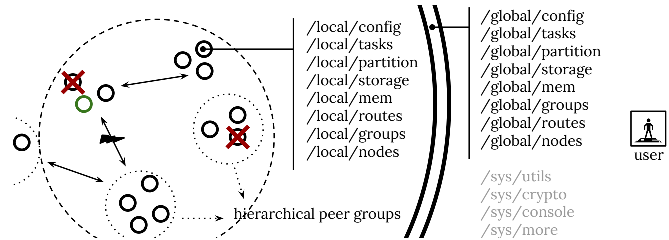

Figure 1: The core of Ar is an extensible service architecture, which is then used to bundle a small set of built-in services. These services are instantiated at the level of node groups (*e.g.*, `local`, `global`).

Using the web as a distributed computing platform, however, requires solving a fundamental problem: unifying individual accessible resources into a useful and coherent programming environment that supports general-purpose computing. Prior attempts [7, 8, 19, 26] were limited by either (i) focusing on *ad hoc* programs [7, 19] that specialize on a particular workload (*e.g.*, prime-number generation), or (ii) requiring significant server-side support [8, 26]. Solving this problem buys us access to Internet-scale resources at the cost of creating a completely new programming infrastructure.

This paper describes Ar, a novel programming environment embedded in JavaScript execution engines that achieves the goal of using the web as a distributed computing platform. Ar *lifts* capabilities locally available by web browser engines (*e.g.*, storage, execution) to their distributed equivalents (Fig. 1). To lift local capabilities, it provides the essential infrastructure for building an extensible architecture of services. This infrastructure is then used to create a small library of built-in services that come bundled with the system and support its critical functions. While Ar is developed (and presented here) directly in JavaScript, its ideas are not language-specific.

The paper is structured as follows: §2 introduces Ar from the perspective of the programmer; §3 presents the design of the service architecture; §4 outlines the prototype implementation and preliminary results; §5 discusses practical challenges and limitations; §6 compares with related work.

2 PROGRAMMER PERSPECTIVE

From the perspective of the programmer, Ar is a stateful service library namespaced under the `ar` object. At the top-level, `ar` exposes a map from groups of nodes to services running on these groups. Services are objects combining internal state with a set of methods. As a concrete example, `ar.cx1.storage.get` will call the `get` method on the `storage` service of the `cx1` group of nodes.

To launch a node, end-users simply visit a URL from their web browser—for example, by following an email link or using a bookmark. The browser fetches and executes system code, effectively

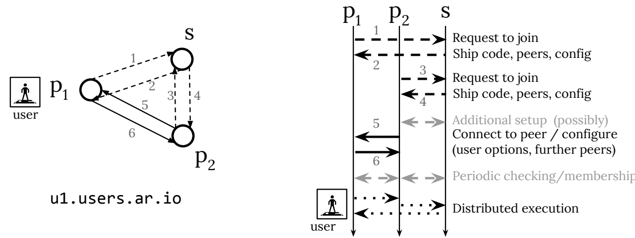

Figure 2: Node startup, from client/server to peer-to-peer.

```
ar.sensors.tasks.exec(() => {
  let update = () => {
    if (ctx.temp > ar.local.mem["max"].temp)
      ar.local.mem["max"] = ctx;
  };
  ar.local.packages.load('clima', (_, clima) => {
    clima.readTemp(clima.CELSIUS, (_, temp) => {
      c = {temp: temp, id: ar.local.info.getNode()};
      ar.desktop.tasks.exec(update, {ctx: c});
    });
  });
});
```

Listing 1: Using Ar to record the maximum temperature across a network of sensor devices.

booting the system, its built-in services, and some built-in application programs such as the interactive shell. As soon as the node has completed booting, it communicates with its peers to register with them, receive updates, and generally interact with the rest of the deployment (Fig 2).

Detailed Example: Sensor Data Aggregation

Consider the problem of recording the highest temperature across a network of sensor nodes. The Ar programmer would write code similar to the one in List. 1, interactively or as part of a script loaded by an HTTP response. Error handling is omitted (by convention, we use underscores to bind `Error` values that will be discarded).

This program loads and executes a function on every node that is part of the `sensors` group (l.1–11). Each node loads a module (l.6) to read the temperature sensor (l.7), and executes a function `update` on `desktop`, a group that contains only a single node (l.9). The `update` function (l.2–5) updates a memory location with the highest temperature recorded.

A few details of the underlying environment are worth noting: (i) first-class support for function distribution, meaning that functions and their context can be transmitted across nodes like any other well-typed value, (ii) a cooperative concurrency model for individual nodes, meaning that code decides when to yield, (iii) an asynchronous, continuation-passing call style that hides the distinction between local and remote execution, and (iv) the ability to dynamically rebind the `ar` object in the context of a function, effectively limiting the incoming code's access to built-in capabilities. Notably, the absence of the first two features would mean that the state update would require a transaction to safeguard against race conditions from competing nodes. In Ar, however, users can send several updates in a single function that is guaranteed to run atomically.

Further Illustrative Examples

Many classes of applications fit well into the Ar model. In each of these three "micro-examples", the user shares a URL similar to the one shown below:

```
z5.kzuse.users.ar.io/CalcSpace?particles=3M
```

Other users can share computing resources by clicking on the URL. The path of the URL is specified with the help of a service responsible for exposing other services. The domain name and other technical details (*e.g.*, NAT hole punching) can be handled by a centralized group of servers (as in our detailed example).

Distributed Proof of Work Users write a procedure that calls a cryptographic hash function using different inputs in an attempt to find a specific pattern (*e.g.*, number of leading zeros). In a semi-centralized case, the node providing the code acts as a controller dividing the search space by the workers. In a more decentralized fashion, each worker subdivides the available search space so that it can recursively hand off chunks to newly-joined, subordinate workers.

Volunteer Computing As a generalization of the previous case, users write a module that donates volunteer resources by periodically contacting peers of a `master` group for receiving jobs and reporting on results. The distributed storage system can be used to read input data (*e.g.*, images) and write results. End-users can choose to expose attenuated capabilities to the underlying resources of their pool by providing a proxy to the Ar object.

Distributed Denial-of-Service Users can setup an DDoS experiment by sharing a procedure that floods a machine with superfluous requests—for example, to explore DoS mitigation in server-like environments [10, 27]. Since Ar running on individual nodes provides a cooperatively scheduled environment, user code needs to plan for periodic (but tiny—*e.g.*, 1ms) back-off to check event queues for incoming coordination messages.

Service Interface

Each service in Ar exposes a set of methods. The argument evaluation strategy is strict (eager) and call-by-value: given an argument, the system will evaluate it, serialize it, and possibly send it to a remote node. There is no distinction between calling a service method and sending a message to a service. Service methods conform to following interface type:

```
operation :: Optional Value ->
             Optional Options ->
             Optional (Error -> [Value] -> ())
             -> ()
```

The first argument represents some state and can be any well-typed value in the programming language (*e.g.*, `exec` in List. 1 is supplied a function).

The second argument express specific quality-of-service requirements including service-specific knobs for tuning inescapable trade-offs [1, 13, 17] in distributed systems. Configurations can be set at a fine granularity—that of individual calls or messages—and are themselves regular objects: users can always group configurations

together, store them in the distributed storage system, assign them to variables, and reuse them on multiple messages for entire applications.

The third argument is an optional continuation function that is called when the operation completes. This function has a special type: it takes two arguments—a possible error and a possible list of results, both of which will be provided by the first function. Error value of `null` signals expected list of results.

Other Abstractions

Node groups and protection capabilities are fundamental system services not detailed in this document; we outline them here.

Node Groups Ar unifies heterogeneous nodes, virtual or physical computing machines of varying capabilities. Nodes implement the abstraction of a computing device (with their own storage, processing *etc.*). In practice, they correspond to a web browser tab which in turn usually corresponds to an operating system process. Users can easily simulate distributed operation on a single physical computer by opening multiple browser tabs.

Nodes are organized into *non-disjoint* sets: the same node can exist in and be addressable through multiple groups. Explicit support for node groups means that groups can provide tailored services based on the capabilities and structure of the underlying nodes as well as the needs of the applications executing atop. Addressing groups of nodes as a single entity is the common pattern, with the possible exception of a *self* singleton group that refers to the current node.

Protection A flexible protection scheme based on sandboxes and object capabilities allows fine-grained access control on resources made available by users. A `sandbox` service can be used to launch incoming scripts into dedicated compartments that, by default, do not provide access to the system's internal structures. Using object capabilities users can attenuate or even rewire access to any functionality available by the system—including built-in and add-on services. This is done by binding custom values to the `ar` name in the context of a sandbox; this capability is further automated by a policy expression language that groups common preset values (*e.g.*, invoking `local`-only storage even when `global` storage is invoked). As a result, users can share computational resources by running tasks from other users on their own infrastructure.

3 SERVICE ARCHITECTURE

This section describes the core internals, an extensible service architecture and how it is used to create a small library of built-in services.

Core System The core of Ar is a routing table. Routing is achieved using a function that takes as an argument a message and passes that message to the service responsible for handling it. A message can encode any well-typed language value, including primitives, functions, and objects. The lower-level internals of the core recognize a couple more types that skip or modify parsing; notable examples include streams of objects, raw-data messages, and raw-data streams.

Messages encoding functions are particularly interesting, as functions provide the necessary support to ship services among nodes. As a result, the aforementioned routing table can be extended during runtime by registering more services. In fact, built-in services are also loaded and instantiated during runtime, often by requesting the majority of their functionality from peer nodes at the edge.

The vast majority of the system's functionality beyond the routing function comes from services. Services are collections of functions along with control-level internal state. These functions can be configured to tune trade-offs related to distribution, scale, and heterogeneity (see `Options` in "Service Interface"); part of the internal state holds the set of available, default, and set parameters.

Service Binding Services are loaded, instantiated, and bound at the level of individual groups. When a node group is asked to bind a new service, Ar starts by locating and loading a service template. It then adds hooks, carefully-placed variable names in scope, for binding values later; for example, it creates a variable named `ar` within the scope of the service. After loading the code, it uses introspection to discover the set of available parameters, their sets of possible values, and documentation associated with both. It populates the default values for all these parameters based on a combination of user inputs—*e.g.*, upon group creation and service instantiation. Finally, it creates the correct `ar` value depending on the capabilities available to the newly instantiated service, and binds it to the `ar` variable.

Messages arriving for services with identical names bound to different node groups will probably get serviced by a different instance of the same service template. This is not certain: as users are free to bind any value they want to a service name, the service might not be an instance of the same template, or it might be the exact same instance. The core routing function described earlier is also exposed as a service instantiated for each node group.

Standard Library Thus far, system services have been described as a more general architecture and associated runtime transformations. Table 1 presents a set of example services that come with Ar and provide foundational functionality. The following paragraphs outline a small set of them; they do not cover the technical details, as these services are replaceable, nor are they intended to be taken as a complete list, as the system is extensible.

One of the most important set of services Ar provides is distributed storage capabilities. A set of services offer the abstraction of a single-dimensional key-value store that partitions data between all nodes in a group. Both in-memory and persistent operations are supported as well as tunable indexing, replication, and consistency guarantees at the level of individual objects. Unispace [28], a multi-dimensional scheme based on hyperspace hashing [12], supports efficient queries on secondary properties. Storage services are used extensively by Ar itself to store and query internal structures. From the point of view of the user, the interface supports a very small number of (four) operations, significantly simplifying state management.

Another important set of services provides first-class support for node and node group management. As described earlier, node groups hold a somewhat central position in Ar for tackling the trade-off between generality, scalability, and accessibility. Using these services, users (and other services) can process topologies and define dynamic overlays that can change at runtime for different phases of a single task. The underlying node list for each node

	Service	Use	Example Call	Representative Options
State	mem	Distributed shared memory	`get({an:"obj"}, {key: "0c3f"}, 1)`	key, replication, consistency
	fs	Persistent key-value storage	`put(obj, {replication: 3})`	— " —, cache
	uni	Persistent multi-dimensional storage	`search({dimensions: ["id"], val: "1"})`	— " —, dimensions, keys
Nodes	nodes	Node information and management	`spawn({number: 3, peer: local.node})`	halt, peer, number, version
	groups	Node group info. and management	`create({name: "primegen", inherit: []})`	name, services, inherit
	service	Service and instantiation infrastructure	`wrap({methods: ["get", "put"]})`	query, constructor, methods
Execution	task	Task management	`exec(pow, {nodes: "first", args: args})`	nodes, replicas, completion
	sandbox	Software-based isolation primitives	`run(f, {id: "s1", ctx: {log: false}})`	globals, context, maxTime
	modules	Module fetch/share	`get("crawler", {source: "local"})`	interface, source, repo
Comm/on	message	Scalable communication primitives	`send({a: "msg"}, {order: false})`	order, consistency, path,
	request	Request-oriented communication	`head("http://up.ar.net", null)`	path, domain, type, stream,
	routes	Bidirectional mapping of names to services	`put(add, {path: "/add"})`	path, access-control
Support	partition	Predicate-based *k*-top nodes	`put(obj, {key: "k", top: 3, type: "any"})`	selection, top, type
	info	Information on underlying infrastructure	`get({keys: /cpu/i, summary: true})`	summary, keys, values
	config	Low-level configuration	`get("version")`	persist
Utility	events	Interrupt event bus	`on("peer-down", (v) =>{v.f()})`	(no callback)
	log	Distributed logging infrastructure	`warn("result", "was", 1 + 2)`	(varargs; no callback)
	doc	Documentation capabilities	`get(/ar.$/, {module: "services"})`	module, code, other

Table 1: Example built-in services; example calls take an asynchronous continuation function as additional argument.

supports time-travelling so that services bound with a group can query the state of Ar in the past. When the topology of a group changes (*e.g.*, node failure), group services send upcalls to interested services, which register the transition and adapt accordingly.

Another group of services is related to distributed code loading and execution. As alluded to, earlier, users can load code dynamically on multiple nodes in the system. Automated runtime transformations are used to address various problems of distribution. For example, users can automatically extract an object capturing the programming interface (API) of a remote program or library; this object hides remote and provides the illusion of local invocation. Users can schedule general purpose programs whose components are distributed functions, ship functions to data, or schedule dependencies between functions executing concurrently. As described in "Service Binding", the system is fully dynamic: there is therefore no notion of installation and any code about to execute can be fetched dynamically. A distributed package manager simplifies fetching code and dealing with dependencies.

Communication services give applications direct access to communication within a group. Applications can register custom own paths in the routing table as inboxes for the delivery of messages. They support various communication semantics (*e.g.*, multicast, anycast, point-to-point) and message reliability options (*e.g.*, at-least-one, ordered, at-most-once).

Support services generally provide foundational support to other services but are rarely used standalone. For example, a map of the underlying node capabilities is used when spawning nodes and rendez-vous partitioning is used in some of the storage services. Several utility services complete the standard library. For example, a group-based event bus provides the ability to emit events and register event handlers.

4 PRELIMINARY EVALUATION

We are in the process of building a prototype implementation of the ideas described in this paper. Here we present preliminary results on a few aspects of the system under development. These are intended to show feasibility and potential, rather than draw final conclusions.

Implementation Services in our prototype are bound statically and ship along with the core of system. They are split between `local` and `global` groups; there is no support for group instantiation yet. The current version of the system is a little over 5K lines of ECMAScript v5.1 (ES5) code. Additionally, there are libraries for supporting the user interface (*e.g.*, interactive shell—1.1K lines of code). We refrained from using ES6 features beyond arrow functions to simplify the serialization and de-serialization infrastructure; fully-supporting ES6 is mostly a matter of engineering effort.

Experiment Setup The majority of the system has been implemented and tested on a portable computer with four 1.1GHz CPUs, 8GB LPDDR3 SDRAM at 1600MHz, and 256GB of PCIe 2.0 (5.0 GT/s) storage. This hardware environment setup is in line with Ar's goal of unifying *commodity* devices at the edge.

For testing operation on the web browser, we use Chrome version 61.0.3163.100; for "headless" operation, we run the system on Node.js [9] bundled with V8 v6.1.534.48, LibUV v1.15.0, and standard library v8.9.3. For setup-related experiments, we host a copy of all software on a server on the same 10MBps network to minimize wide-area inconsistencies, Experiments unrelated to system setup were run on a single machine.

Setup Time In the first experiment, we measure the time required to setup a three-node distributed environment. We launch the system from a web-browser with a freshly-cleared cache and measure various latencies.

For the first node, as Ar is not resident in the web browser's cache, it takes a total 218.93ms to fetch the code and complete the bootup process that includes instantiating services and launching the shell. A total of 92.63ms was required to retrieve the code: 0.19ms for the request, 17.31ms waiting time until the receipt of the first byte, and 75.57ms for content download. It takes another 126.32ms for the startup process to complete, at which point the user has a shell running.

After the startup completion event, the default handler from the startup script asks the system to spawn two new nodes in separate tabs in the background. For these two nodes, and any subsequent ones, the system code is resident in the browser's cache. This lowers fetching and loading times significantly (*i.e.*, by 3-4

orders of magnitude), but adds 92.81ms for spawning a fresh process per tab. The combined total is about half a second (404.55ms).

Memory Footprint Launching Ar almost always starts three nodes. On desktops this results in somwehat generous amounts of memory: the primary node that loads the shell-related modules occupies 18.6MB and the other two 17.3MB. We were able, however, to run an instance of the system with all of its services in a headless browser with under 1MB of memory.

Runtime Performance To get a sense of distributed execution overheads, we developed four trivially-distributed programs— word-count, grep, top-n, and n-grams—and run them using 1MB and 1GB input files. Performance is comparable with Unix pipelines.

We used `perf` sampling (with a sampling rate of 1ms) to better understand the most frequent code-paths taken. 98.94% of the time, the system was found in systemic overheads: 76.18% coming from processing TCP streams (*e.g.*, calls to `__libc_close`) and (18.26%) from system calls (*e.g.*, allocation, spin locks, CSK accepts). Ar's generic `service` template, the prototype object from where all services inherit, was the most heavily invoked function unrelated to HTTP and averaging 1.09% of the samples.

Elasticity To quantify the systemâĂŹs elasticity (responsiveness to to changing workloads) we performed two experiments that spawned 5K virtual nodes on a single machine.

In the first experiment, we launch each node with a startup configuration that runs a single "node-shutdown" command when the bootup process completes. We run this sequentially in a loop where we spawn a node only after the previous node has shutdown. On average, "blinking" an Ar node takes a total of 98ms, most of which is spent in systemic-level overheads (*i.e.*, spawning process, loading source).

In the second experiment, we do not shut down previously spawned nodes. Starting 5K nodes sequentially takes 362.466s (an average of 72.5ms/node). If, however, we spawn 5K nodes in parallel, we get a total of 15.403s (an average of 3ms/node).

5 CHALLENGES AND LIMITATIONS

This section discusses practical challenges and limitations in the development of the system.

Direct vs. CPS Functions that do not perform I/O, such as `Math.add`, follow a direct call style; functions that do I/O, such as `storage.get` follow an asynchronous, continuation-passing style (CPS). This mismatch, inherently present in modern web browsers, is a serious threat to distribution transparency as the calling style depends on the relative location of the function. How can the system transform caller code automatically when a pure function migrates to a remote node and needs to be called in CPS?

Storage Types Web browsers today feature various types of storage; they can be broken down into many different types (*e.g.*, persistent, temporary), but not all of them are available everywhere. To make things more complicated, these different types have drastically different size limits—some of which are configured by the end-user. Low capability with custom runtimes (*e.g.*, Tessel2 [25], Espruinos [23]) do not even offer standardized APIs. How can the system abstract over all these types of storage while notifying the applications?

Node Multiplexing Web browsers break up storage based on *origin*, the name of the domain and sub-domain from where the code was fetched. In our setting, all tabs are served from the same origin. This is rather unusual for web applications: how does the browser know that two different Ar instances need to access different storage?

Value Serialization Ar needs the ability to serialize any well-typed value expressible in JavaScript; however, JSON, JavaScript's native data interchange format, does not support encoding arbitrary language values: cycles within an object are lost, there is no syntax for function literals, and descriptions of properties (*e.g.*, ownership, visibility, writability) stemming directly from the language specification are not captured. How can the system communicate such values, including function closures whose values are bound during runtime?

Security Ar offers improved security to resource-donors as they run distributed applications in a sandboxed environment. However, donors can inspect or alter data and computation from other users. These are known issues in the volunteer-computing literature. Could the system infer information about the intended semantics of distributed applications (*e.g.*, deterministic computation) or offer a library of automated runtime transformations for dealing with these concerns?

6 RELATED WORK

Ar resembles both conventional and distributed operating systems in its explicit statefulness; the state of the programmatic library is distributed across multiple computers [6, 20, 21, 29].

Applications in Ar are not language-agnostic. In language-based distributed systems [2, 5, 15], applications use the same programming language abstractions and bind to the standard library, use programmatic runtime transformations to manipulate state and interfaces, and exploit language-based safety for protection. Similar to Ar, Emerald [5] emphasized prototype-based object mobility and runtime performance of individual nodes. Argus [15] introduced asynchronous interfaces to unify local and remote execution. Ar supports state replication, but lacks Argus' support for transactions. The cooperative concurrency model and first-class support for functions provide partial compensation.

Ar has the additional advantage of runtime extensibility, somewhat like extensible micro-kernels [4], but with built-in introspection [3] and explicit support for service naming [20].

Recent proposals [18, 24] to revisit design decisions behind older distributed operating systems and language runtimes have appeared as consequences of several decades of technology advances; Ar's new approach is building a distributed, language-based operating system for the web. It offers a viable transition path, as it allows users to dispense with the Unix abstractions and POSIX API without necessarily losing all forms of backward-compatibility.

The idea of using overlays to maintain such a delicate balance has been attempted before with systems [5, 11, 15]. Inferno [11] was a very similar approach: it proposed portability across various environments (*e.g.*, standalone and hosted), used a just-in-time compiler as its kernel, and, like Plan 9 [22], used filesystem namespaces similar to the dynamic variable (re-)binding presented in Section 2. However, instead of following Limbo's CPS-inspired

preemptive concurrency model and synchronous channels, Ar supports an Actor-like cooperative concurrency model and asynchronous messages-passing.

Researches have proposed the use of the web for volunteer-computing applications [14, 16], but these applications solve a particular problem or class of problems. Our proposal is somewhat antithetical, in the sense that it aims at a *general-purpose* environment. In such an environment, specialized applications can be used as domain-specific modules.

7 CONCLUSION

This paper proposes using the web as a distributed computing platform at the edge. Our proposed environment, Ar, aims to unify individual accessible resources into a useful and coherent programming environment that supports general-purpose computing. This is achieved by providing the infrastructure for a pluggable service architecture, which is then used to create and bundle a library of built-in services. The resulting system lifts the capabilities locally available by web browser engines to their distributed equivalents.

Acknowledgments We would like to thank the anonymous reviewers for their helpful feedback. This research was funded in part by NSF HALCYON (grant CNS-1513687) and DARPA XD3 (grant DARPA-BAA-15-56). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DARPA.

REFERENCES

[1] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.
[2] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. 1985. The Eden system: A technical review. *IEEE Transactions on Software Engineering* 1 (1985), 43–59.
[3] John K Bennett. 1987. *The design and implementation of distributed Smalltalk*. Vol. 22. ACM.
[4] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. 1995. Extensibility safety and performance in the SPIN operating system. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 267–283.
[5] Andrew P Black, Norman C Hutchinson, Eric Jul, and Henry M Levy. 2007. The development of the Emerald programming language. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, 11–1. http://www.emeraldprogramminglanguage.org/authorsVersion.pdf
[6] David Cheriton. 1988. The V distributed system. *Commun. ACM* 31, 3 (1988), 314–333. http://www.eecs.berkeley.edu/~prabal/resources/osprelim/Che88.pdf
[7] Andrew Collins. 2010. Distributed Pi. (2010). http://cgi.csc.liv.ac.uk/~acollins/pi Accessed: 2017-06-11.
[8] Reginald Cushing, Ganeshwara Herawan Hananda Putra, Spiros Koulouzis, Adam Belloum, Marian Bubak, and Cees De Laat. 2013. Distributed computing on an ensemble of browsers. *IEEE Internet Computing* 17, 5 (2013), 54–61.
[9] Ryan Dahl and the Node.js Foundation. 2009. Node.js. (2009). https://nodejs.org Accessed: 2017-06-11.
[10] Henri Maxime Demoulin, Tavish Vaidya, Isaac Pedisich, Nik Sultana, Bowen Wang, Jingyu Qian, Yuankai Zhang, Ang Chen, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. 2017. A Demonstration of the DeDoS Platform for Defusing Asymmetric DDoS Attacks in Data Centers. In *Proceedings of the SIGCOMM Posters and Demos (SIGCOMM Posters and Demos '17)*. ACM, New York, NY, USA, 71–73. https://doi.org/10.1145/3123878.3131990
[11] Sean M Dorward, Rob Pike, David Leo Presotto, Dennis M Ritchie, Howard W Trickey, and Philip Winterbottom. 1997. The Inferno operating system. *Bell Labs Technical Journal* 2, 1 (1997), 5–18. http://www.vitanuova.com/inferno/papers/bltj.pdf
[12] Robert Escriva, Bernard Wong, and Emin Gün Sirer. 2012. HyperDex: A Distributed, Searchable Key-value Store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '12)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2342356.2342360
[13] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* 33, 2 (2002), 51–59.
[14] Jik-Soo Kim, Peter Keleher, Michael Marsh, Bobby Bhattacharjee, and Alan Sussman. 2007. Using content-addressable networks for load balancing in desktop grids. In *Proceedings of the 16th international symposium on High performance distributed computing*. ACM, 189–198.
[15] Barbara Liskov. 1988. Distributed programming in Argus. *Commun. ACM* 31, 3 (1988), 300–312. https://people.csail.mit.edu/alinush/6.824-spring-2015/papers/argus88.pdf
[16] Virginia Lo, Daniel Zappala, Dayi Zhou, Yuhong Liu, and Shanyu Zhao. 2004. Cluster computing on the fly: P2P scheduling of idle cycles in the internet. In *International Workshop on Peer-to-Peer Systems*. Springer, 227–236.
[17] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-Optimal Read-Only Transactions.. In *OSDI*. 135–150.
[18] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. 2014. The Case for the Holistic Language Runtime System. In *Proceedings of the 1st International Workshop on Rack-scale Computing*. http://research.microsoft.com/en-us/events/wrsc2014/maas14case.pdf
[19] Edward Meeds, Remco Hendriks, Said Al Faraby, Magiel Bruntink, and Max Welling. 2015. MLitB: machine learning in the browser. *PeerJ Computer Science* 1 (2015), e11.
[20] Sape J Mullender, Guido Van Rossum, AS Tanenbaum, Robbert Van Renesse, and Hans Van Staveren. 1990. Amoeba: A distributed operating system for the 1990s. *Computer* 23, 5 (1990), 44–53. https://www.cs.cornell.edu/home/rvr/papers/Amoeba1990s.pdf
[21] John K Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. 1988. The Sprite network operating system. *Computer* 21, 2 (1988), 23–36. http://www.research.ibm.com/people/f/fdouglis/papers/sprite.pdf
[22] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, et al. 1990. Plan 9 from Bell Labs. In *Proceedings of the summer 1990 UKUUG Conference*. 1–9. http://css.csail.mit.edu/6.824/2014/papers/plan9.pdf
[23] Ltd. Pur3. 2017. Espruino. (2017). http://www.espruino.com/ Accessed: 2017-06-11.
[24] Malte Schwarzkopf, Matthew P Grosvenor, and Steven Hand. 2013. New wine in old skins: the case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*. ACM, 9. http://www.cl.cam.ac.uk/~ms705/pub/papers/2013-apsys-dios.pdf
[25] The Tessel Foundation. 2013. Tessel 2. (2013). https://tessel.io Accessed: 2017-06-11.
[26] Tai-Lun Tseng, Shih-Hao Hung, and Chia-Heng Tu. 2015. Migratom.js: A JavaScript migration framework for distributed Web computing and mobile devices. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM, 798–801.
[27] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization. In *Networked and Distributed Systems Security (NDSS'18)*. https://doi.org/10.14722/ndss.2018.23131
[28] Nikos Vasilakis, Yash Palkhiwala, and Jonathan M. Smith. 2017. Query-efficient Partitions for Dynamic Data. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*. ACM, New York, NY, USA, Article 23, 8 pages. https://doi.org/10.1145/3124680.3124744
[29] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. 1983. The LOCUS distributed operating system. In *ACM SIGOPS Operating Systems Review*, Vol. 17. Acm, 49–70. http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.344