

***Na Kika*: Secure Service Execution and Composition in an Open Edge-Side Computing Network**

Robert Grimm, Guy Lichtman, Nikolaos Michalakis,
Amos Elliston, Adam Kravetz, Jonathan Miller, and Sajid Raza
New York University

Abstract

Making the internet’s edge easily extensible fosters collaboration and innovation on web-based applications, but also raises the problem of how to secure the execution platform. This paper presents *Na Kika*, an edge-side computing network, that addresses this tension between extensibility and security; it safely opens the internet’s edge to all content producers and consumers. First, *Na Kika* expresses services as scripts, which are selected through predicates on HTTP messages and composed with each other into a pipeline of content processing steps. Second, *Na Kika* isolates individual scripts from each other and, instead of enforcing inflexible a-priori quotas, limits resource consumption based on overall system congestion. Third, *Na Kika* expresses security policies through the same predicates as regular application functionality, with the result that policies are as easily extensible as hosted code and that enforcement is an integral aspect of content processing. Additionally, *Na Kika* leverages a structured overlay network to support cooperative caching and incremental deployment with low administrative overhead.

1 Introduction

Web-based applications increasingly rely on the dynamic creation and transformation of content [5]. Scaling such applications to large and often global audiences requires placing them close to clients, at the edge of the internet. Edge-side content management provides the CPU power and network bandwidth necessary to meet the needs of local clients. As a result, it reduces load on origin servers, bandwidth consumption across the internet, and latency for clients. It also absorbs load spikes, e.g., the Slashdot effect, for underprovisioned servers. Based on similar observations, commercial content distribution networks (CDNs) already offer edge-side hosting services. For example, Akamai hosts customer-supplied J2EE components on edge-side application servers [1]. Furthermore, many ISPs provide value-added services, such as “web accelerators”, by dynamically transforming web content on the edge. However, commercial CDNs and ISPs have limited reach. To manage the trust necessary for exposing their hosting infrastructure to other people’s code, they rely on traditional, contract-based business relationships. As a result, commercial CDNs

and ISPs are ill-suited to collaborative and community-based development efforts; they best serve as amplifiers of (large) organizations’ web servers.

At the same time, many community-based efforts are exploring the use of web-based collaboration to address large-scale societal and educational problems. For instance, researchers at several medical schools, including New York University’s, are moving towards web-based education [10, 43, 45] to address nationally recognized problems in medical education [28, 49]. The basic idea is to organize content along narrative lines to re-establish context missing in clinical practice, complement textual presentation with movies and animations to better illustrate medical conditions and procedures, and leverage electronic annotations (post-it notes) and discussions for building a community of students and practitioners. Furthermore, such web-based educational environments dynamically adapt content to meet students’ learning needs and transcode it to enable ubiquitous access, independent of devices and networks. A crucial challenge for these efforts is how to combine the content and services created by several groups and organizations into a seamless learning environment and then scale that environment to not only the 67,000 medical students in the U.S., but also the 850,000 physicians in the field as well as to medical personnel in other countries facing similar problems.

Taking a cue from peer-to-peer CDNs for static content, such as CoDeeN [47, 48] and Coral [13], *Na Kika*¹ targets cooperative efforts that do not (necessarily) have the organizational structure or financial resources to contract with a commercial CDN or cluster operator and seeks to provide an edge-side computing network that is fully open: Anyone can contribute nodes and bandwidth to *Na Kika*, host their applications on it, and access content through it. In other words, by opening up the internet’s edge, *Na Kika* seeks to provide the technological basis for improved collaboration and innovation on large-scale web-based applications. In this paper, we explore how *Na Kika* addresses the central challenge raised by such an open architecture: how to secure our execution platform while also making it easily extensible.

Na Kika, similar to other CDNs, mediates all HTTP in-

¹Our system is named after the octopus god of the Gilbert Islands, who put his many arms to good use during the great earth construction project.

teractions between clients and servers through edge-side proxies. Also similar to other CDNs, individual edge-side nodes coordinate with each other to cache content, through a structured overlay in our case. *Na Kika*'s key technical difference—and our primary contribution—is that both hosted applications and security policies are expressed as scripted event handlers, which are selected through predicates on HTTP messages and composed into a pipeline of content processing stages. Our architecture builds on the fact that HTTP messages contain considerable information about clients, servers, and content to expose the same high-level language for expressing functionality and policies alike—with the result that policies are as easily extensible as hosted code and that enforcement is an integral aspect of content processing. A second difference and contribution is that *Na Kika*'s resource controls do not rely on a-priori quotas, which are too inflexible for an open system hosting arbitrary services with varying resource requirements. Instead *Na Kika* limits resource consumption based on congestion: If a node's resources are overutilized, our architecture first throttles requests proportionally to their resource consumption and eventually terminates the largest resource consumers.

Our use of scripting and overlay networks provides several important benefits. First, scripting provides a uniform and flexible mechanism for expressing application logic and security policies alike. Second, scripting simplifies the task of securing our edge-side computing network, as we can more easily control a small execution engine and a small number of carefully selected platform libraries than restricting a general-purpose computing platform [20, 41]. Third, scripting facilitates an API with low cognitive complexity: *Na Kika*'s event-based API is not only easy to use but, more importantly, already familiar to programmers versed in web development. Fourth, the overlay ensures that *Na Kika* is incrementally scalable and deployable. In particular, the overlay supports the addition of nodes with minimal administrative overhead. It also helps with absorbing load spikes for individual sites, since one cached copy (of either static content or service code) is sufficient for avoiding origin server accesses.

At the same time, *Na Kika* does have limitations. Notably, it is unsuitable for applications that need to process large databases, as the databases need to be moved to the internet's edge as well. Furthermore, since *Na Kika* exposes all functionality as scripts, applications whose code needs to be secret cannot utilize it (though obfuscation can help). Next, by utilizing *Na Kika*, content producers gain capacity but also give up control over their sites' performance. We expect that any deployment of our edge-side computing network is regularly monitored to identify persistent overload conditions and to

rectify them by adding more nodes. Finally, while *Na Kika* protects against untrusted application code, it does trust edge-side nodes to correctly cache data and execute scripts. As a result, it is currently limited to deployments across organizations that can be trusted to properly administer local *Na Kika* nodes. We return to this issue in Section 6.

2 Related Work

Due to its palpable benefits, several projects have been exploring edge-side content management. A majority of these efforts, such as ACDN [33], ColTrES [8], Tuxedo [38], vMatrix [2], and IBM's WebSphere Edge Server [17] (which is used by Akamai), explore how to structure the edge-side hosting environment. Since they are targeted at closed and trusted deployments, they do not provide an extension model, nor do they include the security and resource controls necessary for hosting untrusted code. In contrast, the OPES architecture for edge-side services recognizes the need for extensibility and service composition [4, 23]. While it does not specify how composition should be achieved, OPES does define potential security threats [3]. Their scope and magnitude is illustrated by experiences with the CoDeeN open content distribution network [48].

Next, Active Cache [9] and SDT [19] enable content processing in proxy caches. While they do not provide an extension mechanism, they do provide precise control over edge-side processing through server-specified HTTP headers. Furthermore, while SDT enforces only coarse-grained resource controls for Perl and none for Java, Active Cache executes Java code with resource limits proportional to the size of the content being processed. Unlike these systems, Pai et al.'s proxy API [31] provides fine-grained extensibility for web proxies through an event-based API akin to ours. At the same time, their work focuses on enabling high-performance extensions in trusted deployments, while our work focuses on containing arbitrary extensions in untrusted deployments. Finally, Active Names [46] are explicitly designed for extensibility and service composition, chaining processing steps in a manner comparable to *Na Kika*'s scripting pipeline. In fact, by introducing a new naming interface, Active Names offer more flexibility for content processing than our work. However, they also require a new service infrastructure, while *Na Kika* integrates with the existing web.

While cooperative caching has its limitations [50], coordination between edge-side nodes is still important for scaling a system, in particular to balance load and absorb load spikes. To this end, CoDeeN [47], ColTrES [8], and Tuxedo [38] are exploring the use of domain-specific topologies and algorithms. In contrast, *Na Kika* leverages previous work on structured overlay

networks [13, 16, 35, 42, 52] for coordinating between local caches. We believe that structured overlays provide a robust and scalable alternative to domain-specific coordination strategies. Additionally, structured overlays have already been used successfully for caching static content [13, 18].

In most edge-side systems, nodes cannot be entrusted with the sole copies of application data, and hard state requiring stronger consistency than the web’s expiration-based guarantees (or lack thereof) must remain on origin servers. In contrast, ACDN [33] reduces access latency for such data by replicating it across edge nodes and by providing full serializability through a primary replica. Gao et al. [14] explore alternative replication strategies by exposing a set of distributed objects that make different trade-offs between consistency, performance, and availability. Alternatively, the continuous consistency model provides a framework for expressing such trade-offs through a uniform interface to hard state [51]. *Na Kika*’s support for application state builds on Gao et al.’s approach, with the primary difference that replicated state is subject to *Na Kika*’s security and resource controls.

Web content processing is (obviously) not limited to edge nodes and can be performed on servers and clients as well. For example, *Na Kika* has several similarities with the cluster-based TACC architecture [12]. Both *Na Kika* and TACC rely on a pipeline of programs that process web content, and both build on the expiration-based consistency model of the web to cache both original and processed content. *Na Kika* differs in that it targets proxies distributed across the wide area and thus needs to carefully contain hosted code. Comparable to *Na Kika*, DotSlash [53] helps absorb load spikes by moving script execution to other servers in a “mutual-aid community”. Unlike *Na Kika*, it has no extension model and does not provide security and resource controls. At the other end, client side includes [34] (CSI) move the assembly of dynamic content to the client, which can improve latency for clients relying on low bandwidth links. However, due to their focus on assembling content fragments, CSI are not suitable for content processing in general. The underlying edge side includes [29, 44] (ESI) can easily be supported within *Na Kika*.

Finally, based on the realization that system security can clearly benefit from a dedicated and concise specification of policies, a considerable number of efforts have explored policy specification languages. For example, domain and type enforcement [7], XACML [22], and trust management systems such as PolicyMaker, KeyNote, and SPKI [6, 11] include languages for expressing and enforcing policies. All these systems require explicitly programmed calls to the respective reference monitor. In contrast, previous work on security

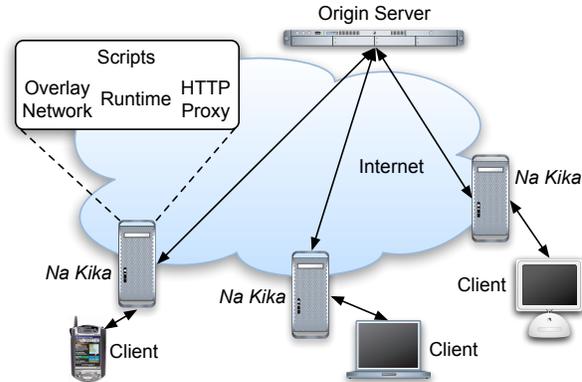


Figure 1: Illustration *Na Kika*’s architecture. Edge-side proxies mediate all HTTP interactions between clients and servers by executing scripts; proxies also coordinate with each other through an overlay network.

for extensible systems advocates the separation of policies, enforcement, and functionality and relies on binary interposition to inject access control operations into executing code [15, 39]. The WebGuard policy language relies on a similar approach for securing web-based applications [40]. Since *Na Kika*’s programming model is already based on interposition, we leverage the same predicate selection mechanism for application logic and policies, thus eliminating the need for a separate policy specification language.

3 Architecture

Like other extensions to the basic web infrastructure and as illustrated in Figure 1, *Na Kika* relies on proxies that mediate HTTP interactions between clients and servers. To utilize these proxies, content producers and consumers need to change existing web practices along two lines. First, content producers need to publish the necessary edge-side processing scripts on their web sites. Content producers need not provide scripts for an entire site at once. Rather, they can transition to *Na Kika* piecemeal, starting with content whose creation or transformation exerts the highest resource demands on their servers. Second, links need to be changed by appending “.nakika.net” to a URL’s hostname, so that *Na Kika*’s name servers can redirect clients to (nearby) edge nodes. As described in [13], URLs can be modified by content publishers, third parties linking to other sites, as well as by users. Furthermore, URLs can be rewritten through a service in our architecture. While *Na Kika* also supports static proxy configuration in browsers, we prefer URL rewriting as it allows for more fine-grained load balancing between edge nodes and presents a uniform, location-independent interface for using our architecture.

3.1 Programming Model

The functionality of hosted services and applications is specified through two *event handlers*, which are written in JavaScript. Our architecture does not depend on the choice of scripting language and could support several languages. We chose JavaScript because it already is widely used by web developers. Additionally, we found its C-like syntax and prototype-based object model helpful in writing scripts quickly and with little code; though we had to add support for byte arrays to avoid unnecessarily copying data. The `onRequest` event handler accepts an HTTP request and returns either a request for continued processing or a response representing the corresponding content or error condition. The `onResponse` event handler accepts an HTTP response and always returns a response. A pair of `onRequest` and `onResponse` event handlers mimics the high-level organization of any HTTP proxy and represents the unit of composition in *Na Kika*: a *scripting pipeline stage*.

In providing two interposition points for HTTP processing, *Na Kika* differs from other systems, such as Active Cache [9], SDT [19], and TACC [12], which only interpose on HTTP responses. Interposition on requests is necessary for HTTP redirection and, more importantly, as a first-line defense for enforcing access controls. It also is more efficient if responses are created from scratch, as it avoids accessing a resource before edge-side processing. To facilitate the secure composition of untrusted services, *Na Kika* relies on fewer event handlers than Pai et al.'s proxy API [31]; though it does provide similar expressivity, notably, to control the proxy cache, through its platform libraries.

Similar to ASP.NET and JSP, requests and responses are not passed as explicit arguments and return values, but are represented as global JavaScript objects. Using global objects provides a uniform model for accessing functionality and data, since native-code libraries, which we call *vocabularies*, also expose their functionality through global JavaScript objects. *Na Kika* provides vocabularies for managing HTTP messages and state and for performing common content processing steps. In particular, it provides support for accessing URL components, cookies, and the proxy cache, fetching other web resources, managing hard state, processing regular expressions, parsing and transforming XML documents, and transcoding images. We expect to add vocabularies for performing cryptographic operations and transcoding movies as well. Figure 2 illustrates an example `onResponse` event handler.

For HTTP responses, the body always represents the entire instance [25] of the HTTP resource, so that the resource can be correctly transcoded [19]. If the response represents an unmodified or partial resource, it is instantiated, for example, by retrieving it from the cache, when

```
onResponse = function() {
  var buff = null, body = new ByteArray();
  while (buff = Response.read()) {
    body.append(buff);
  }

  var type = ImageTransformer.
    type(Response.contentType);
  var dim = ImageTransformer.
    dimensions(body, type);
  if (dim.x > 176 || dim.y > 208) {
    var img;
    if (dim.x/176 > dim.y/208) {
      img = ImageTransformer.transform(body,
        type, "jpeg", 176, dim.y/dim.x*208);
    } else {
      img = ImageTransformer.transform(body,
        type, "jpeg", dim.x/dim.y*176, 208);
    }
    Response.setHeader("Content-Type",
      "image/jpeg");
    Response.setHeader("Content-Length",
      img.length);
    Response.write(img);
  }
}
```

Figure 2: An example `onResponse` event handler, which transcodes images to fit onto the 176 by 208 pixel screen of a Nokia cell phone. It relies on the image transformer vocabulary to do the actual transcoding. The response body is accessed in chunks to enable cut-through routing; though the transformer vocabulary does not yet support it, with the script buffering the entire body.

a script accesses the body.

Event Handler Selection

To provide script modularity and make individual pipeline stages easily modifiable, stages do not consist of a fixed pair of event handlers; rather, the particular event handlers to be executed for each stage are selected from a collection of event handlers. To facilitate this selection process, pairs of `onRequest` and `onResponse` event handlers are associated with predicates on HTTP requests, including, for example, the client's IP address or the resource's URL. Conceptually, *Na Kika* first evaluates all of a stage's predicates and then selects the pair with the closest valid match for execution.

The association between event handlers and predicates is expressed in JavaScript by instantiating *policy objects*. As illustrated in Figure 3, each policy object has several properties that contain a list of allowable values for the corresponding HTTP message fields. Each policy object also has two properties for the `onRequest` and `onResponse` event handlers and an optional `nextStages` property for scheduling additional stages as discussed below. Lists of allowable values support prefixes for URLs, CIDR notation for IP

```

p          = new Policy();
p.url     = [ "med.nyu.edu",
             "medschool.pitt.edu" ];
p.client  = [ "nyu.edu",
             "pitt.edu" ];
p.onResponse = function() { ... }
p.register();

```

Figure 3: An example policy object. The policy applies the `onResponse` event handler to all content on servers at NYU’s or University of Pittsburgh’s medical schools accessed from within the two universities. The call to `register()` activates the policy.

addresses, and regular expressions for arbitrary HTTP headers. When determining the closest valid match, different values in a property’s list are treated as a disjunction, different properties in a policy object are treated as a conjunction, and null properties are treated as truth values. Furthermore, precedence is given to resource URLs, followed by client addresses, then HTTP methods, and finally arbitrary headers. Null event handlers are treated as no-ops for event handler execution, thus making it possible to process only requests or responses or to use a stage solely for scheduling other stages.

Selecting event handlers by declaring predicates on HTTP messages avoids long sequences of if-else statements in a single, top-level event handler, thus resulting in more modular event processing code. When compared to the additional HTTP headers used by Active Cache and SDT for selecting edge-side code, predicate-based script selection also enables the interposition of code not specified by the origin server, an essential requirement for both composing services and enforcing security. While designing *Na Kika*, we did consider a domain-specific language (DSL) for associating predicates with event handlers instead of using JavaScript-based policy objects. While a DSL can be more expressive (for example, by allowing disjunction between properties), we rejected this option because it adds too much complexity—both for web developers targeting *Na Kika* and for implementors of our architecture—while providing little additional benefits. We also considered performing predicate selection on HTTP responses, but believe that pairing event handlers results in a simpler programming model, with little loss of expressivity. Also matching responses requires a very simple change to our implementation.

Scripting Pipeline Composition

By default, each scripting pipeline has three stages. The first stage provides administrative control over clients’ access to our edge-side computing network. It can, for example, perform rate limiting, redirect requests, or reject them altogether. The second stage performs site-specific processing, which typically serves as a surrogate

```

procedure EXECUTE-PIPELINE(request)
  forward ← EMPTY
  backward ← EMPTY
  ▷ Start with administrative control and site-specific stages
  PUSH(forward, "http://nakika.net/serverwall.js")
  PUSH(forward, SITE(request.url) + "/nakika.js")
  PUSH(forward, "http://nakika.net/clientwall.js")
  repeat
    ▷ Schedule stages and execute onRequest
    script ← FETCH-AND-EXECUTE(POP(forward))
    policy ← FIND-CLOSEST-MATCH(script, request)
    PUSH(backward, policy)
    if policy.onRequest ≠ NIL then
      response ← RUN(policy.onRequest, request)
      ▷ If handler creates response, reverse direction
      if response ≠ NIL then exit repeat end if
    end if
    if policy.nextStages ≠ NIL then
      ▷ Add new stages
      PREPEND(forward, policy.nextStages)
    end if
  until forward = EMPTY
  if response = NIL then
    ▷ Fetch original resource
    response ← FETCH(request)
  end if
  repeat
    ▷ Execute onResponse
    policy ← POP(backward)
    if policy.onResponse ≠ NIL then
      RUN(policy.onResponse, response)
    end if
  until backward = EMPTY
  return response
end procedure

```

Figure 4: Algorithm for executing a pipeline. The algorithm interleaves computing a pipeline’s schedule with `onRequest` event handler execution, so that matching can take into an account when an event handler modifies the request, notably to redirect it.

for the origin server and actually creates dynamic content. For example, this stage adapts medical content in a web-based educational environment to a students’ learning needs. The third stage provides administrative control over hosted scripts’ access to web resources. Similar to the first stage, it can redirect or reject requests.

To perform additional processing, each pipeline stage can dynamically schedule further stages by listing the corresponding scripts in a policy object’s `nextStages` property. As shown in Figure 4, the dynamically scheduled stages are placed directly after the scheduling stage but before other, already scheduled stages. A site-specific script can thus delay content creation until a later, dynamically scheduled stage, while also scheduling additional processing before that stage. Examples for such intermediate services include providing annotations (electronic post-it notes) for textual content and transcoding movies for access from mobile devices. To put it differently, each site can configure its own pipeline and thus has full control over how its content is created and transformed—within the bounds of *Na Kika*’s administrative control. At the same time, new services,

such as visualization of the spread of diseases, can easily be layered on top of existing services, such as geographical mapping, even when the services are provided by different sites: the new service simply adjusts the request, including the URL, and then schedules the original service after itself. Both services are executed within a single pipeline on the same *Na Kika* node.

The scripts for each stage are named through regular URLs, accessed through regular HTTP, and subject to regular HTTP caching. As shown in Figure 4, the administrative control scripts are accessed from well-known locations; though administrators of *Na Kika* nodes may override these defaults to enforce their own, location-specific security controls. Site-specific scripts are accessed relative to the server's domain, in a file named `nakika.js`, which is comparable to the use of `robots.txt` and `favicon.ico` for controlling web spiders and browser icons, respectively. All other services, that is, dynamically scheduled pipeline stages, can be hosted at any web location and are accessed through their respective URLs.

In combining content creation with content transformation, our architecture's scripting pipeline is reminiscent of the Apache web server and Java servlets. At the same time, both Apache and Java servlets have a more complicated structure. They first process a request through a chain of input filters, then create a response in a dedicated module (the content handler for Apache and the actual servlet for Java servlets), and finally process the response through a chain of output filters. In mirroring an HTTP proxy's high-level organization, *Na Kika*'s scripting pipeline stages have a simpler interface—requiring only two event handlers—and are also more flexible, as any `onRequest` event handler can generate a response. Furthermore, the content processing pipelines for Apache and Java servlets can only be configured by code outside the pipelines, while each stage in *Na Kika*'s scripting pipelines can locally schedule additional stages—with the overall result that *Na Kika* is more flexible and more easily extensible, even in the presence of untrusted code.

Na Kika Pages

While our architecture's event-based programming model is simple and flexible, a large portion of dynamic content on the web is created by markup-based content management systems, such as PHP, JSP, and ASP.NET. To support web developers versed in these technologies, *Na Kika* includes an alternative programming model for site-specific content. Under this model, HTTP resources with the `nkp` extension or `text/nkp` MIME type are subject to edge-side processing: all text between the `<?nkp` start and `?>` end tags is treated as JavaScript and replaced by the output of running that code. These

```
bmj = "bmj.bmjournals.com/cgi/reprint";
nejm = "content.nejm.org/cgi/reprint";
p = new Policy();
p.url = [ bmj, nejm ];

p.onRequest = function() {
  if (! System.isLocal(Request.clientIP)) {
    Request.terminate(401);
  }
}
p.register();
```

Figure 5: An example policy that prevents access to the digital libraries of the BMJ (British Medical Journal) and the New England Journal of Medicine from clients outside a *Na Kika* node's hosting organization. The 401 HTTP error code indicates an unauthorized access.

so-called *Na Kika Pages* are implemented on top of *Na Kika*'s event-based programming model through a simple, 60 line script. We expect to utilize a similar technique to also support edge side includes [29, 44] (ESI) within the *Na Kika* architecture.

3.2 Security and Resource Controls

Na Kika's security and resource controls need to protect (1) the proxies in our edge-side computing network against client-initiated exploits, such as those encountered by CoDeeN [48], (2) the proxies against exploits launched by hosted code, and (3) other web servers against exploits carried through our architecture. We address these three classes of threats through admission control by the client-side administrative control stage, resource controls for hosted code, and emission control by the server-side administrative control stage, respectively. Of course, it is desirable to drop requests early, before resources have been expended [26], and, consequently, requests that are known to cause violations of *Na Kika*'s security and resource controls should always be rejected at the client-side administrative control stage.

Because the two administrative control stages mediate all HTTP requests and responses entering and leaving the system, they can perform access control based on client and server names as well as rate limiting based on request rates and response sizes. The corresponding policies are specified as regular scripts and can thus leverage the full expressivity of *Na Kika*'s predicate matching. For instance, Figure 5 shows a policy object rejecting unauthorized accesses to digital libraries, which is one type of exploit encountered by CoDeeN. For more flexibility, security policies can also leverage dynamically scheduled stages. For example, the two administrative control stages can delegate content blocking to separate stages whose code, in turn, is dynamically created by a script based on a blacklist.

To enforce resource controls, a resource manager

```

procedure CONTROL(resource)
  priorityq ← EMPTY
  if IS-CONGESTED(resource) then ▷ Track usage and throttle
    for site in ACTIVE-SITES() do
      UPDATE(site.usage, resource)
      ENQUEUE(priorityq, site)
      THROTTLE(site, resource)
    end for
  else if ¬ IS-RENEWABLE(resource) then ▷ Track usage
    for site in ACTIVE-SITES() do
      UPDATE(site.usage, resource)
    end for
  end if
  WAIT(TIMEOUT) ▷ Let throttling take effect
  if IS-CONGESTED(resource) then
    TERMINATE(DEQUEUE(priorityq)) ▷ Kill top offender
  else
    UNTHROTTLE(resource) ▷ Restore normal operation
  end if
end procedure

```

Figure 6: Algorithm for congestion control. The CONTROL procedure is periodically executed for each tracked resource. Note that our implementation does not block but rather polls to detect timeouts.

tracks CPU, memory, and bandwidth consumption as well as running time and total bytes transferred for each site’s pipelines. It also tracks overall consumption for the entire node. As shown in Figure 6, if any of these resources is overutilized, the resource manager starts throttling requests proportionally to a site’s contribution to congestion and, if congestion persists, terminates the pipelines of the largest contributors. A site’s contribution to congestion captures the portion of resources consumed by its pipelines. For renewable resources, i.e., CPU, memory, and bandwidth, only consumption under overutilization is included. For nonrenewable resources, i.e., running time and total bytes transferred, all consumption is included. In either case, the actual value is the weighted average of past and present consumption and is exposed to scripts—thus allowing scripts to adapt to system congestion and recover from past penalization.

To complete resource controls, all pipelines are fully sandboxed. They are isolated from each other, running, for example, with their own heaps, and can only access select platform functionality. In particular, all regular operating system services, such as processes, files, or sockets, are inaccessible. The only resources besides computing power and memory accessible by scripts are the services provided by *Na Kika*’s vocabularies (that is, native-code libraries).

We believe that *Na Kika*’s congestion-based resource management model is more appropriate for open systems than more conventional quota-based resource controls for two reasons. First, open systems such as *Na Kika* have a different usage model than more conven-

tional hosting platforms: they are open to all content producers and consumers, with hosting organizations effectively donating their resources to the public. In other words, services and applications should be able to consume as many resources as they require—as long as they do not interfere with other services, i.e., cause congestion. Second, quota-based resource controls require an administrative decision as to what resource utilization is legitimate. However, even when quotas are set relative to content size [9], it is hard to determine appropriate constants, as the resource requirements may vary widely. We did consider setting fine-grained quotas through predicates on HTTP messages, comparable to how our architecture selects event handlers. However, while predicate-based policy selection is flexible, it also amplifies the administrative problem of which constants to choose for which code.

Our architecture’s utilization of scripting has two advantages for security and resource control when compared to other edge-side systems. First, administrative control scripts simplify the development and deployment of security policy updates. Once a fix to a newly discovered exploit or abuse has been implemented, the updated scripts are simply published on the *Na Kika* web site and automatically installed across all nodes when cached copies of the old scripts expire. In contrast, CoDeeN and other edge-side systems that hard code security policies require redistribution of the system binaries across all nodes. Though *Na Kika* still requires binary redistribution to fix security holes in native code. Second, providing assurance that hosted services and applications are effectively secured is simpler for scripts than for Java or native code. Our starting point is a bare scripting engine to which we selectively add functionality, through vocabularies, rather than trying to restrict a general purpose platform after the fact.

3.3 Hard State

The web’s expiration-based consistency model for cached state is sufficient to support a range of edge-side applications, including content assembly (through, for example, edge-side includes [29, 44]) or the transcoding of multi-media content. However, a complete platform for edge-side content management also requires support for managing hard state such as edge-side access logs and replicated application state. Edge-side logging provides accurate usage statistics to content producers, while edge-side replication avoids accessing the origin server for every data item.

Na Kika performs access logging on a per-site basis. Logging is triggered through a site’s script, which specifies the URL for posting log updates. Periodically, each *Na Kika* node scans its log, collects all entries for each specific site, and posts those portions of the log to the

specified URLs.

Na Kika's support for edge-side replication builds on Gao et al.'s use of distributed objects, which, internally, rely on domain-specific replication strategies to synchronize state updates and support both pessimistic and optimistic replication [14]. Like Gao et al., *Na Kika*'s hard state replication relies on a database for local storage and a reliable messaging service for propagating updates, which are exposed through vocabularies. Unlike Gao et al., *Na Kika*'s hard state replication is implemented by regular scripts. Updates are accepted by a script, written to local storage, and then propagated to other nodes through the messaging layer. Upon receipt of a message on another node, a regular script processes the message and applies the update to that node's local storage. As a result, *Na Kika* provides content producers with considerable flexibility in implementing their domain-specific replication strategies. For example, the script accepting updates can propagate them only to the origin server to ensure serializability or to all nodes to maximize availability. Furthermore, the script accepting messages can easily implement domain-specific conflict resolution strategies. To secure replicated state, *Na Kika* partitions hard state amongst sites and enforces resource constraints on persistent storage. Since update processing is performed by regular scripts, it already is subject to *Na Kika*'s security and resource controls.

3.4 Overlay Network

The *Na Kika* architecture relies on a structured overlay network for coordinating local caches and for enabling incremental deployment with low administrative overhead. From an architectural viewpoint, the overlay is treated largely as a black box, to be provided by an existing DHT [13, 16, 35, 42, 52]. This reflects a conscious design decision on our end and provides us with a test case for whether DHTs can, in fact, serve as robust and scalable building blocks for a global-scale distributed system. Our prototype implementation builds on Coral [13], which is well-suited to the needs of our architecture, as Coral explicitly targets soft state and includes optional support for DNS redirection to local nodes. As we deploy *Na Kika*, we expect to revisit the functionality provided by the DHT. Notably, load balancing, which is currently provided at the DNS level, can likely benefit from application-specific knowledge, such as the number of concurrent HTTP exchanges being processed by a node's scripting pipelines.

3.5 Summary

The *Na Kika* architecture leverages scripting and overlay networks to provide an open edge-side computing network. First, *Na Kika* exposes a programming model already familiar to web developers by organizing hosted

services and applications into a pipeline of scripted event handlers that process HTTP requests and responses. Second, it provides a secure execution platform by mediating all HTTP processing under administrative control, by isolating scripts from each other, and by limiting resource utilization based on overall system congestion. Third, it provides extensibility by dynamically scheduling event handlers within a pipeline stage as well as additional pipeline stages through predicate matching. Finally, it provides scalability by organizing all nodes into an automatically configured overlay network, which supports the redirection of clients to (nearby) edge nodes and the addition of new nodes with low administrative overhead.

At the same time, web integration is not entirely complete, as URLs need to be rewritten for *Na Kika* access. As already discussed, URLs can be automatically rewritten by web browsers, hosted code, as well as servers and, consequently, the need for manual rewriting will decrease over time. Furthermore, while our architecture protects against client- and script-initiated exploits, it does not currently protect against misbehaving edge-side nodes. In particular, nodes can arbitrarily modify cached content, which is especially problematic for administrative control scripts. We return to the issue of content integrity in Section 6.

4 Implementation

Our prototype implementation of *Na Kika* builds on three open source packages: the Apache 2.0 web server, the Mozilla project's SpiderMonkey JavaScript engine [27], and the Coral distributed hashtable [13]. We chose Apache for HTTP processing because it represents a mature and cross-platform web server. Similarly, SpiderMonkey is a mature and cross-platform implementation of JavaScript and used across the Mozilla project's web browsers. Additionally, our prototype includes a preliminary implementation of hard state replication, which relies on the Java-based JORAM messaging service [30] and exposes a vocabulary for managing user registrations, as required by the SPECweb99 benchmark. Our implementation adds approximately 23,000 lines of C code to the 263,000 lines of code in Apache, the 123,000 lines in SpiderMonkey, and the 60,000 lines in Coral. The majority of changes is to Apache and mostly contained in Apache modules. Our modified Apache binary, including dynamically loaded libraries, is 10.6 MByte large and the Coral DHT server is 13 MByte.

As already mentioned in Section 3.1, Apache structures HTTP processing into a chain of input filters that operate on requests, followed by a content handler that generates responses, followed by a chain of output filters that operate on responses. Our prototype implements the scripting pipeline by breaking each stage

into a pair of input and output filters, which execute the `onRequest` and `onResponse` event handlers, respectively, and by dynamically inserting the pair into Apache’s filter chain. The content handler is a modified version of Apache’s `mod_proxy`, which implements the proxy cache and, in our version, also interfaces with the DHT. If an `onRequest` event handler generates an HTTP response, our implementation sets a flag that prevents the execution of scripts in later pipeline stages and of the proxy caching code, while still conforming with Apache’s sequencing of input filters, content handler, and output filters.

To provide isolation, our implementation executes each pipeline in its own process and each script, in turn, in its own user-level thread and with its own scripting context, including heap. Scripting contexts are reused to amortize the overhead of context creation across several event handler executions; this is safe because JavaScript programs cannot forge pointers and the heap is automatically garbage collected. A separate monitoring process tracks each pipeline’s CPU, memory, and network consumption and periodically executes the congestion control algorithm in Figure 6. To throttle a site’s pipelines, the monitoring process sets a flag in shared memory, which causes the regular Apache processes to reject requests for that site’s content with a server busy error. To terminate a site’s pipelines, the monitoring process kills the corresponding Apache processes, thus putting an immediate stop to processing even if a pipeline is executing a vocabulary’s native code.

Employing per-script user-level threads also helps integrate script execution with Apache, while still exposing a simple programming model. In particular, Apache’s sequence of input filters, content handler, and output filters is not necessarily invoked on complete HTTP requests and responses. Rather, each filter is invoked on chunks of data, the so-called *bucket brigades*, as that data becomes available. As a result, Apache may interleave the execution of several `onRequest` and `onResponse` event handlers. Per-script user-level threads hide this piecemeal HTTP processing from script developers, providing the illusion of scripts running to completion before invoking the next stage. To avoid copying data between Apache and the scripting engine, our implementation adds byte arrays as a new core data type to SpiderMonkey. Whenever possible, these byte arrays directly reference the corresponding bucket brigade buffers.

The policy matching code trades off space for dynamic predicate evaluation performance. While loading a script and registering policy objects, the matcher builds a decision tree for that pipeline stage, with nodes in the tree representing choices. Starting from the root of the tree, the nodes represent the components of a resource URL’s server name, the port, the components of the path, the

Name	Description
Proxy	A regular Apache proxy.
DHT	The proxy with an integrated DHT.
Admin	A <i>Na Kika</i> node evaluating one matching predicate and executing empty event handlers for each of the two administrative control stages.
Pred- n	The Admin configuration plus another stage evaluating predicates for n policy objects, with no matches.
Match-1	The Admin configuration plus another stage evaluating one matching predicate and executing the corresponding, empty event handlers.

Table 1: The different micro-benchmark configurations.

components of the client address, the HTTP methods, and, finally, individual headers. If a property of a policy object does not contain any values, the corresponding nodes are skipped. Furthermore, if a property contains multiple values, nodes are added along multiple paths. When all properties have been added to the decision tree, the event handlers are added to the current nodes, once for each path. With the decision tree in place, dynamic predicate evaluation simply is a depth-first search across the tree for the node closest to the leaves that also references an appropriate event handler. Decision trees are cached in a dedicated in-memory cache. The implementation also caches the fact that a site does *not* publish a policy script, thus avoiding repeated checks for the `nakika.js` resource.

5 Evaluation

To evaluate *Na Kika*, we performed a set of local micro-benchmarks and a set of end-to-end experiments, which include wide area experiments on the PlanetLab distributed testbed [32]. The micro-benchmarks characterize (1) the overhead introduced by *Na Kika*’s DHT and scripting pipeline and (2) the effectiveness of our congestion-based resource controls. The end-to-end experiments characterize the performance and scalability of a real-world application and of a modified SPECweb99 benchmark. We also implemented three new services to characterize the extensibility of our edge-side computing network. In summary, our experimental results show that, even though the scripting pipeline introduces noticeable overheads, *Na Kika* is an effective substrate both for scaling web-based applications and for extending them with new functionality.

5.1 Micro-Benchmarks

To characterize the overheads introduced by *Na Kika*’s DHT and scripting pipeline, we compare the performance of a *Na Kika* node with a regular Apache proxy cache for accessing a single, static 2,096 byte document representing Google’s home page (without in-

Configuration	Cold Cache	Warm Cache
Proxy	3	1
DHT	5	1
Admin	16	2
Pred-0	19	2
Pred-1	20	2
Match-1	21	2
Pred-10	22	2
Pred-50	30	2
Pred-100	41	2

Table 2: Latency in milliseconds for accessing a static page under the different configurations.

line images). Since static resources are already well-served by existing proxy caches and CDNs, these micro-benchmarks represent a worst-case usage scenario for *Na Kika*. After all, any time spent in the DHT or in the scripting pipeline adds unnecessary overhead. For all experiments, we measured the total time of a client accessing the static web page through a proxy—with client, proxy, and server being located on the same, switched 100 Mbit ethernet. The proxy runs on a Linux PC with a 2.8 GHz Intel Pentium 4 and 1 GB of RAM.

We performed 18 experiments, representing 9 different configurations under both a cold and a warm proxy cache. The different configurations are summarized in Table 1 and determine the overhead of DHT integration, baseline administrative control, predicate matching, and event handler invocation, respectively. For the cold cache case of the *Admin*, *Pred-n*, and *Match-1* configurations, the administrative control and site-specific scripts are fetched from the local server and evaluated to produce the corresponding decision tree. For the warm cache case, the cached decision tree is used. Resource control is disabled for these experiments.

Table 2 shows the latency in milliseconds for the 18 different experiments. Each number is the average of 10 individual measurements. Overall, the results clearly illustrate the basic cost of utilizing *Na Kika*: its scripting pipeline. For the *Pred-n* and *Match-1* configurations under a cold cache, loading the actual page takes 2.9 ms and loading the script takes between 2.5 ms and 5.6 ms, depending on size. Additionally, the creation of a scripting context takes 1.5 ms. Finally, parsing and executing the script file takes between 0.08 ms and 17.8 ms, again, depending on size. However, the results also illustrate that our implementation’s use of caching—for resources, scripting contexts, and decision trees—is effective. Retrieving a resource from Apache’s cache takes 1.1 ms and retrieving a decision tree from the in-memory cache takes 4 μ s. Re-using a scripting context takes 3 μ s. Finally, predicate evaluation takes less than 38 μ s for all configurations. However, these operations also result in a higher CPU load: the *Na Kika*

node reaches capacity with 30 load-generating clients at 294 requests/second (rps) under *Match-1*, while the plain Apache proxy reaches capacity with 90 clients at 603 rps on the same hardware. Since both resources and scripts only need to be accessed when reaching their expiration times, we expect that real world performance is closer to warm cache than cold cache results. Furthermore, most web resources are considerably larger than Google’s home page, so that network transfer times will dominate scripting pipeline latency.

Resource Controls

To characterize the effectiveness of *Na Kika*’s congestion-based resource management, we compare the performance of a *Na Kika* node with and without resource controls under high load, such as that caused by a flash crowd. For these experiments, the *Na Kika* proxy runs on the same Linux PC as before. Load is generated by accessing the same 2,096 byte page under the *Match-1* configuration in a tight loop. With 30 load generators (i.e., at the proxy’s capacity), we measure 294 rps without and 396 rps with resource controls. With 90 load generators (i.e., under overload), we measure 229 rps without and 356 rps with resource controls. If we also add one instance of a misbehaving script, which consumes all available memory by repeatedly doubling a string, the throughput with 30 load generators drops to 47 rps without but only 382 rps with resource controls. For all experiments, the runs with resource controls reject less than 0.55% of all offered requests due to throttling and drop less than 0.08% due to termination, including the one triggering the misbehaving script. These results illustrate the benefits of *Na Kika*’s resource controls. Even though resource management is reactive, throttling is effective at ensuring that admitted requests have sufficient resources to run to completion, and termination is effective at isolating the regular load from the misbehaving one.

5.2 Web-based Medical Education

To evaluate a real-world application running on *Na Kika*, we compare the Surgical Interactive Multimedia Modules [43] (SIMMs) in their original single-server configuration with an initial port to our edge-side computing network. The SIMMs are a web-based educational environment that is being developed by NYU’s medical school. Each SIMM focuses on a particular medical condition and covers the complete workup of a patient from presentation to treatment to follow-up. It consists of rich-media enhanced lectures, annotated imaging studies, pathology data, and animated and real-life surgical footage—comprising around 1 GB of multimedia content per module. The five existing SIMMs already are an integral part of the curriculum at NYU’s medical school and are also used at four other medical schools in

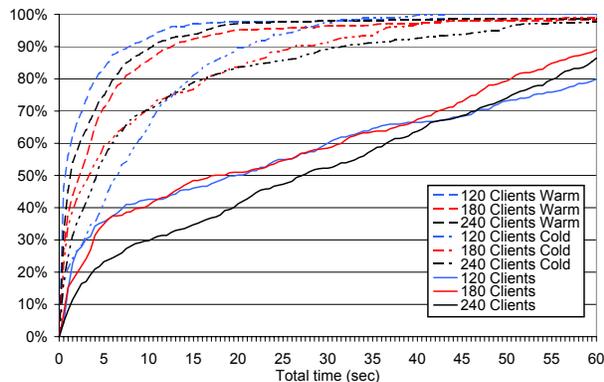


Figure 7: Cumulative distribution function (CDF) for latency to access HTML content in the SIMMs’ single-server and *Na Kika* cold and warm cache configurations.

the U.S. and Australia, with more institutions to follow. The SIMMs rely heavily on personalized and multimedia content but do not contain any real patient data (with its correspondent privacy requirements), thus making them good candidates for deployment on *Na Kika*.

The SIMMs are implemented on top of Apache Tomcat 5.0 and MySQL 4.1. They utilize JSP and Java servlets to customize content for each student as well as to track her progress through the material and the results of sectional assessments. To facilitate future interface changes as well as different user interfaces, customized content is represented as XML and, before being returned to the client, rendered as HTML by an XSL stylesheet (which is the same for all students)². The initial *Na Kika* port off-loads the distribution of multimedia content, since it is large, and the (generic) rendering of XML to HTML, since it is processor intensive, to our edge-side computing network. Content personalization is still performed by the central server; we expect that future versions will also move personalization to the edge.

The port was performed by one of the main developers of the SIMMs and took two days. The developer spent four hours on the actual port—which entailed changing URLs to utilize *Na Kika*, making XML and XSL content accessible over the web, and switching from cookies to URL-based session identifiers as well as from HTTP POSTs to GETs—and the rest of the two days debugging the port. In fact, the main impediment to a faster port was the relative lack of debugging tools for our prototype implementation. The port adds 65 lines of code to the existing code base of 1,900 lines, changes 25 lines, and removes 40 lines. The new `nakika.js` policy consists of 100 lines of JavaScript code.

To evaluate end-to-end performance, we compare the

²An earlier version relied on a custom-built Macromedia Director client for rendering XML. It was abandoned in favor of regular web browsers due to the extra effort of maintaining a dedicated client.

single-server version with the *Na Kika* port accessed through a single, local proxy—which lets us compare baseline performance—and with the *Na Kika* port running on proxies distributed across the wide area—which lets us compare scalability. For all experiments, we measure the total time to access HTML content—which represents client-perceived latency—and the average bandwidth when accessing multimedia files—which determines whether playback is uninterrupted. Load is generated by replaying access logs for the SIMMs collected by NYU’s medical school; log replay is accelerated 4× to produce noticeable activity. For the local experiments, we rely on four load-generating nodes. For the wide-area experiments, 12 load-generating PlanetLab nodes are distributed across the U.S. East Coast, West Coast, and Asia—thus simulating a geographically diverse student population—and, for *Na Kika*, matched with nearby proxy nodes. For *Na Kika*, we direct clients to randomly chosen, but close-by proxies from a preconfigured list of node locations. For the local experiments, the origin server is the same PC as used in Section 5.1; for the wide-area experiments, it is a PlanetLab node in New York.

The local experiments show that, under a cold cache and heavy load, the performance of the single *Na Kika* proxy trails that of the single server. Notably, for 160 clients (i.e., 40 instances of the log replay program running on each of 4 machines), the 90th percentile latency for accessing HTML content is 904 ms for the single server and 964 ms for the *Na Kika* proxy. The fraction of accesses to multimedia content consistently seeing a bandwidth of at least 140 Kbps—the SIMMs’ video bitrate—is 100% for both configurations. However, when adding an artificial network delay of 80 ms and bandwidth cap of 8 Mbps between the server on one side and the proxy and clients on the other side (to simulate a wide-area network), the single *Na Kika* proxy already outperforms the single server, illustrating the advantages of placing proxies close to clients. For 160 clients, the 90th percentile latency for HTML content is 8.88 s for the single server and 1.21 s for the *Na Kika* proxy. Furthermore, only 26.2% of clients see sufficient bandwidth for accessing video content for the single server, while 99.9% do for the *Na Kika* proxy. As illustrated in Figure 7, the advantages of our edge-side computing network become more pronounced for the wide-area experiments. For 240 clients (i.e., 20 programs running on each of 12 machines), the 90th percentile latency for accessing HTML content is 60.1 s for the single server, 31.6 s for *Na Kika* with a cold cache, and 9.7 s with a warm cache. For the single server, the fraction of clients seeing sufficient video bandwidth is 0% and the video failure rate is 60.0%. For *Na Kika* with a cold cache, the fraction is 11.5% and the failure rate is 5.6%. With a warm cache, the fraction is 80.3% and

the failure rate is 1.9%. For *Na Kika*, accesses to multimedia content benefit to a greater extent from a warm cache than accesses to HTML, since PlanetLab limits the bandwidth available to each hosted project.

5.3 Hard State Replication

To further evaluate end-to-end performance in the wide area, we compare the performance of a single Apache PHP server and the same server supported by *Na Kika* running a modified version of the SPECweb99 benchmark. For this experiment, we re-implemented SPECweb99's server-side scripts in PHP and *Na Kika Pages*. The single-server version relies on PHP because it is the most popular add-on for creating dynamic content to the most popular web server [36, 37]. The *Na Kika* version relies on replicated hard state to manage SPECweb99's user registrations and profiles. With clients and five *Na Kika* nodes on the U.S. West Coast and the server located on the East Coast, 80% dynamic requests, 160 simultaneous connections, and a runtime of 20 minutes, the PHP server has a mean response time of 13.7 s and a throughput of 10.8 rps. With a cold cache, the *Na Kika* version has a response time of 4.3 s and a throughput of 34.3 rps. Additional experiments show that the results are very sensitive to PlanetLab CPU load, thus indicating that *Na Kika*'s main benefit for these experiments is the additional CPU capacity under heavy load (and, conversely, that *Na Kika* requires ample CPU resources to be effective). Our SPECweb99 compliance score is 0 due to the limited bandwidth available between PlanetLab nodes. Nonetheless, this benchmark shows that *Na Kika* can effectively scale a complex workload that includes static content, dynamic content, and distributed updates.

5.4 Extensions

To evaluate *Na Kika*'s extensibility, we implemented three extensions in addition to the *Na Kika Pages* extension discussed in Section 3.1: electronic annotations for the SIMMs, image transcoding for small devices, and content blocking based on blacklists. As described below, our experiences with these extensions confirm that *Na Kika* is, in fact, easily extensible. In particular, they illustrate the utility of predicate-based event handler selection and dynamically scheduled pipeline stages. Furthermore, they illustrate that developers can build useful extensions quickly, even if they are not familiar with *Na Kika* or JavaScript programming.

Our first extension adds electronic annotations, i.e., post-it notes, to the SIMMs, thus providing another layer of personalization to this web-based educational environment. The extended SIMMs are hosted by a site outside NYU's medical school and utilize dynamically scheduled pipeline stages to incorporate the *Na Kika* version

of the SIMMs. The new functionality supports electronic annotations by injecting the corresponding dynamic HTML into the SIMMs' HTML content. It also rewrites request URLs to refer to the original content and URLs embedded in HTML to refer to itself, thus interposing itself onto the SIMMs. The resulting pipeline has three non-administrative stages, one each for URL rewriting, annotations, and the SIMMs. The annotations themselves are stored on the site hosting the extended version. This extension took one developer 5 hours to write and debug and comprises 50 lines of code; it leverages a previously developed implementation of electronic annotations, which comprises 180 lines of code.

In contrast to the extension for electronic annotations, which represents one site building on another site's service, our second extension represents a service to be published on the web for use by the larger community. This extension scales images to fit on the screen of a Nokia cell phone and generalizes the `onResponse` event handler shown in Figure 2 to cache transformed content. The extension can easily be modified to support other types and brands of small devices by (1) parameterizing the event handler's screen size and (2) adding new policy objects that match other devices' `User-Agent` HTTP headers. This extension took a novice JavaScript developer less than two hours to write and debug and comprises 80 lines of code.

Our third extension does not provide new functionality, but rather extends *Na Kika*'s security policy with the ability to block sites based on blacklists. Its intended use is to deny access to illegal content through *Na Kika*. The extension is implemented through two dynamically scheduled pipeline stages. The first new stage relies on a static script to dynamically generate the JavaScript code for the second new stage, which, in turn, blocks access to the URLs appearing on the blacklist. The static script reads the blacklist from a preconfigured URL and then generates a policy object for each URL appearing on that blacklist. The `onRequest` event handler for all policy objects is the same handler, denying access as illustrated in Figure 5. This extension took 4.5 hours to write and debug, with an additional 1.5 hours for setting up a testbed. Since this extension represents the developer's first *Na Kika* as well as JavaScript code, the 4.5 hours include one hour mostly spent familiarizing himself with JavaScript. The extension comprises 70 lines of code.

6 Discussion and Future Work

As presented in this paper, *Na Kika* assumes that edge-side nodes are trusted, which effectively limits the organizations participating in a deployment. To allow *Na Kika* to scale to a larger number of edge networks and nodes, we are currently working towards eliminating this requirement by automatically ensuring the integrity

of content served by our edge-side computing network. Content integrity is important for producers and consumers so that, for example, the results of medical studies cannot be falsified. It also is important for the operation of the network itself, as scripts, including those used for administrative control, are accessed through and cached within *Na Kika*.

For original content, protecting against inadvertent or malicious modification reduces to detecting such changes and then retrieving the authoritative version from the origin server. However, using cryptographic hashes, for example, through self-certifying pathnames [24] as suggested in [13], is insufficient, as they cannot ensure freshness. To provide both integrity and freshness, we have already implemented an alternative solution that integrates with HTTP's cache control by adding two new headers to HTTP responses. The `X-Content-SHA256` header specifies a cryptographic hash of the content for integrity and, to reduce load, can be precomputed. The `X-Signature` header specifies a signature over the content hash and the cache control headers for freshness. Our solution requires the use of absolute cache expiration times instead of the relative times introduced in HTTP/1.1 [21] as nodes cannot be trusted to correctly decrement relative times.

For processed or generated content, content integrity cannot be established through hashes and signatures alone, as content processing is performed by potentially untrusted nodes. Instead, we are exploring a probabilistic verification model. Under this model, a trusted registry maintains *Na Kika* membership. To detect misbehaving nodes, clients forward a fraction of content received from *Na Kika* proxies to other proxies, which then repeat any processing themselves. If the two versions do not match, the original proxy is reported to the registry, which uses this information to evict misbehaving nodes from the edge-side computing network.

7 Conclusions

Edge-side content management reduces load on origin servers, bandwidth consumption across the internet, and latency for clients. It also absorbs load spikes for under-provisioned servers. To make these benefits available to *all* content producers and consumers and thus to foster collaboration and innovation on web-based applications, *Na Kika* provides an open architecture for edge-side content creation, transformation, and caching.

Services and applications hosted by *Na Kika* are expressed through scripted event handlers. Event handlers are selected through predicates on HTTP messages and are composed into a pipeline that combines administrative control and site-specific processing. The resulting programming model is not only familiar to web developers versed in client-side scripting and the content pro-

cessing pipelines of Apache and Java servlets, but it also is more secure and more easily extensible. To provide security, *Na Kika*'s scripting pipeline mediates all requests and responses passing through the system. Furthermore, all hosted services and applications are isolated from each other and the underlying operating system and subject to congestion-based resource management: hosted code can consume resources without restriction as long as it does not cause overutilization. To provide incremental scalability, all *Na Kika* nodes are organized into a structured overlay network, which enables DNS redirection of clients to nearby nodes and cooperative caching of both original and processed content. The experimental evaluation demonstrates that *Na Kika*'s prototype implementation is effective at reducing load on origin servers and latency for clients, supporting significantly larger user populations than a single dynamic web server. It also demonstrates that *Na Kika* is, in fact, easily programmable and extensible. 🏠

Acknowledgments

Bill Holloway ported the SIMMs to *Na Kika* and Jake Aviles helped with their evaluation. Robert Soule implemented static content integrity and the security policy extension. Our shepherd, Emin Gün Sirer, and the anonymous reviewers provided valuable feedback on earlier versions of this paper. This material is based in part upon work supported by the National Science Foundation under Grant No. 0537252 and by the New York Software Industry Association.

References

- [1] Akamai Technologies, Inc. A developer's guide to on-demand distributed computing, Mar. 2004.
- [2] A. Awadallah and M. Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proc. 7th IWCW*, Aug. 2002.
- [3] A. Barbir, O. Batuner, B. Srinivas, M. Hofmann, and H. Orman. Security threats and risks for open pluggable edge services (OPES). RFC 3837, IETF, Aug. 2004.
- [4] A. Barbir, R. Penno, R. Chen, M. Hofmann, and H. Orman. An architecture for open pluggable edge services (OPES). RFC 3835, IETF, Aug. 2004.
- [5] L. Bent, M. Rabinovich, G. M. Voelker, and Z. Xiao. Characterization of a large web site population with implications for content delivery. In *Proc. 13th WWW*, pp. 522–533, May 2004.
- [6] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. In J. Vitek and C. D. Jensen, eds., *Secure Internet Programming*, vol. 1603 of *LNCS*, pp. 185–210. Springer, 1999.
- [7] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proc. 17th NCSC*, pp. 18–27, 1985.
- [8] C. Canali, V. Cardellini, M. Colajanni, R. Lancellotti, and P. S. Yu. Cooperative architectures and algorithms for discovery and transcoding of multi-version content. In *Proc. 8th IWCW*, Sept. 2003.
- [9] P. Cao, J. Zhang, and K. Beach. Active Cache: Caching dynamic contents on the web. In *Proc. Middleware '98*, pp. 373–388, Sept. 1998.

- [10] D. M. D'Alessandro, T. E. Lewis, and M. P. D'Alessandro. A pediatric digital storytelling system for third year medical students: The virtual pediatric patients. *BMC Medical Education*, 4(10), July 2004.
- [11] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, IETF, Sept. 1999.
- [12] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proc. 16th SOSISP*, pp. 78–91, Oct. 1997.
- [13] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. 1st NSDI*, Mar. 2004.
- [14] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific data replication for edge services. In *Proc. 12th WWW*, pp. 449–460, May 2003.
- [15] R. Grimm and B. N. Bershad. Separating access control policy, enforcement and functionality in extensible systems. *ACM TOCS*, 19(1):36–70, Feb. 2001.
- [16] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proc. 2nd IPTPS*, vol. 2735 of *LNCS*, pp. 160–169. Springer, Feb. 2003.
- [17] IBM Corporation. *WebSphere Edge Server Administration Guide*. 3rd edition, Dec. 2001.
- [18] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. 21st PODC*, pp. 213–222, July 2002.
- [19] B. Knutsson, H. Lu, J. Mogul, and B. Hopkins. Architecture and performance of server-directed transcoding. *ACM TOIT*, 3(4):392–424, Nov. 2003.
- [20] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. AGENT TCL: Targeting the needs of mobile computers. *IEEE Internet Computing*, 1(4):58–67, Jul./Aug. 1997.
- [21] B. Krishnamurthy, J. C. Mogul, and D. M. Kristol. Key differences between HTTP/1.0 and HTTP/1.1. In *Proc. 8th WWW*, May 1999.
- [22] M. Lorch, S. Proctor, R. Lepro, D. Kafura, and S. Shah. First experiences using XACML for access control in distributed systems. In *Proc. 2003 XMLSEC*, pp. 25–37, 2003.
- [23] W.-Y. Ma, B. Shen, and J. Brassil. Content services network: The architecture and protocols. In *Proc. 6th IWCW*, June 2001.
- [24] D. Mazières and M. F. Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *Proc. 8th SIGOPS Europ. Workshop*, pp. 118–125, Sept. 1998.
- [25] J. C. Mogul. Clarifying the fundamentals of HTTP. In *Proc. 11th WWW*, pp. 25–36, May 2002.
- [26] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive live-lock in an interrupt-driven kernel. *ACM TOCS*, 15(3):217–252, Aug. 1997.
- [27] Mozilla Foundation. SpiderMonkey (JavaScript-C) engine. <http://www.mozilla.org/js/spidermonkey/>.
- [28] National Center for Postsecondary Improvement. Beyond dead reckoning: Research priorities for redirecting American higher education. <http://www.stanford.edu/group/ncpi/documents/pdfs/beyond.dead.reckoning.pdf>, Oct. 2002.
- [29] M. Nottingham and X. Liu. Edge architecture specification, 2001. http://www.esi.org/architecture-spec_1-0.html.
- [30] ObjectWeb. JORAM. <http://joram.objectweb.org/>.
- [31] V. S. Pai, A. L. Cox, V. S. Pai, and W. Zwaenepoel. A flexible and efficient application programming interface for a customizable proxy cache. In *Proc. 4th USITS*, Mar. 2003.
- [32] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Proc. 1st HotNets*, Oct. 2002.
- [33] M. Rabinovich, Z. Xiao, and A. Aggarwal. Computing on the edge: A platform for replicating internet applications. In *Proc. 8th IWCW*, Sept. 2003.
- [34] M. Rabinovich, Z. Xiao, F. Dougliis, and C. Kalmanek. Moving edge-side includes to the real edge—the clients. In *Proc. 4th USITS*, Mar. 2003.
- [35] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware '01*, pp. 329–350, Nov. 2001.
- [36] Security Space. Apache module report. http://www.securityspace.com/s_survey/data/man.200501/apachemods.html, Feb. 2005.
- [37] Security Space. Web server survey. http://www.securityspace.com/s_survey/data/200501/index.html, Feb. 2005.
- [38] W. Shi, K. Shah, Y. Mao, and V. Chaudhary. Tuxedo: A peer-to-peer caching system. In *Proc. 2003 PDPTA*, pp. 981–987, June 2003.
- [39] E. G. Sirer, R. Grimm, A. J. Gregory, and B. N. Bershad. Design and implementation of a distributed virtual machine for networked computers. In *Proc. 17th SOSISP*, pp. 202–216, Dec. 1999.
- [40] E. G. Sirer and K. Wang. An access control language for web services. In *Proc. 7th SACMAT*, pp. 23–30, June 2002.
- [41] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public internet measurement facility. In *Proc. 4th USITS*, pp. 225–238, Mar. 2003.
- [42] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. 2001 SIGCOMM*, pp. 149–160, Aug. 2001.
- [43] M. M. Triola, M. A. Hopkins, M. J. Weiner, W. Holloway, R. I. Levin, M. S. Nachbar, and T. S. Riles. Surgical interactive multimedia modules: A novel, non-browser based architecture for medical education. In *Proc. 17th CBMS*, pp. 423–427, June 2004.
- [44] M. Tsimelzon, B. Weihl, and L. Jacobs. ESI language specification 1.0, 2001. http://www.esi.org/language-spec_1-0.html.
- [45] S. H. J. Uijtdehaage, S. E. Dennis, and C. Candler. A web-based database for sharing educational multimedia within and among medical schools. *Academic Medicine*, 76:543–544, 2001.
- [46] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible location and transport of wide-area resources. In *Proc. 2nd USITS*, pp. 151–164, Oct. 1999.
- [47] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *Proc. 5th OSDI*, pp. 345–360, Dec. 2002.
- [48] L. Wang, K. Park, R. Pang, V. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *Proc. 2004 USENIX*, pp. 171–184, June 2004.
- [49] S. A. Wartman. Research in medical education: The challenge for the next decade. *Academic Medicine*, 69(8):608–614, 1994.
- [50] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. Karlin, and H. M. Levy. On the scale and performance of cooperative web proxy caching. In *Proc. 17th SOSISP*, pp. 16–31, Dec. 1999.
- [51] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proc. 4th OSDI*, pp. 305–318, Oct. 2000.
- [52] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J-SAC*, 22(1):41–53, Jan. 2004.
- [53] W. Zhao and H. Schulzrinne. DotSlash: Providing dynamic scalability to web applications with on-demand distributed query result caching. Tech. Report CUCS-035-05, Columbia University, Sept. 2005.