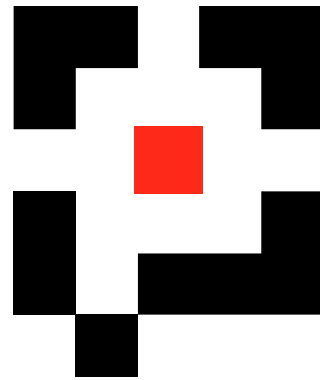


I  
N  
D  
I  
A  
N  
A  
  
U  
N  
I  
V  
E  
R  
S  
I  
T  
Y

Presentation by ANML  
June 2004



pervasive **technology** labs

AT INDIANA UNIVERSITY

IPv6: DNS and Programming

# About the Presenter

---

- Mark Meiss
- Academic Background:
  - B.S. Mathematics, B.S. Computer Science
  - Ph.D. student in Department of Computer Science
- Research interests:
  - Structural analysis of network traffic data
  - High-performance file transfer protocols
  - Autonomous information retrieval agents



# About the Presenter

---

- Professional Experience:
  - Over 10 years in software development
  - With IU IT Services since 1997
  - Worked with Bloomington NOC
  - First employee of ANML
  - Developed Animated Traffic Map, Router Proxy, Tsunami file transfer protocol, etc.



# Overview

---

1. DNS nameserver changes under IPv6
  - New resource types and compatibility issues
2. DNS resolver changes under IPv6
  - Reasons for change and using the new interface



# DNS Nameserver Changes

---



pervasivet<sup>technology</sup>labs  
AT INDIANA UNIVERSITY

# Review of DNS Operation

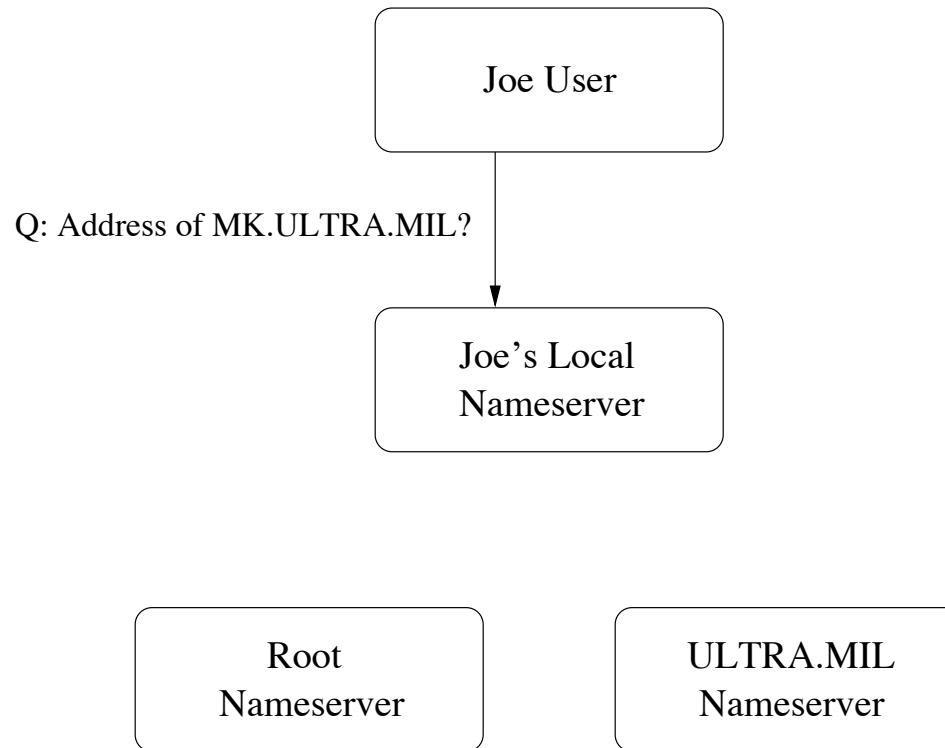
---

- Before we dig too deep, let's review DNS
- The Domain Name Service is a distributed database that maps from *hostname* to *IP address* and vice-versa
  - Includes a variety of record and query types
  - We will focus on the most common queries:
    - Hostname-to-Address (“forward lookup”)
    - Address-to-Hostname (“reverse/inverse lookup”)



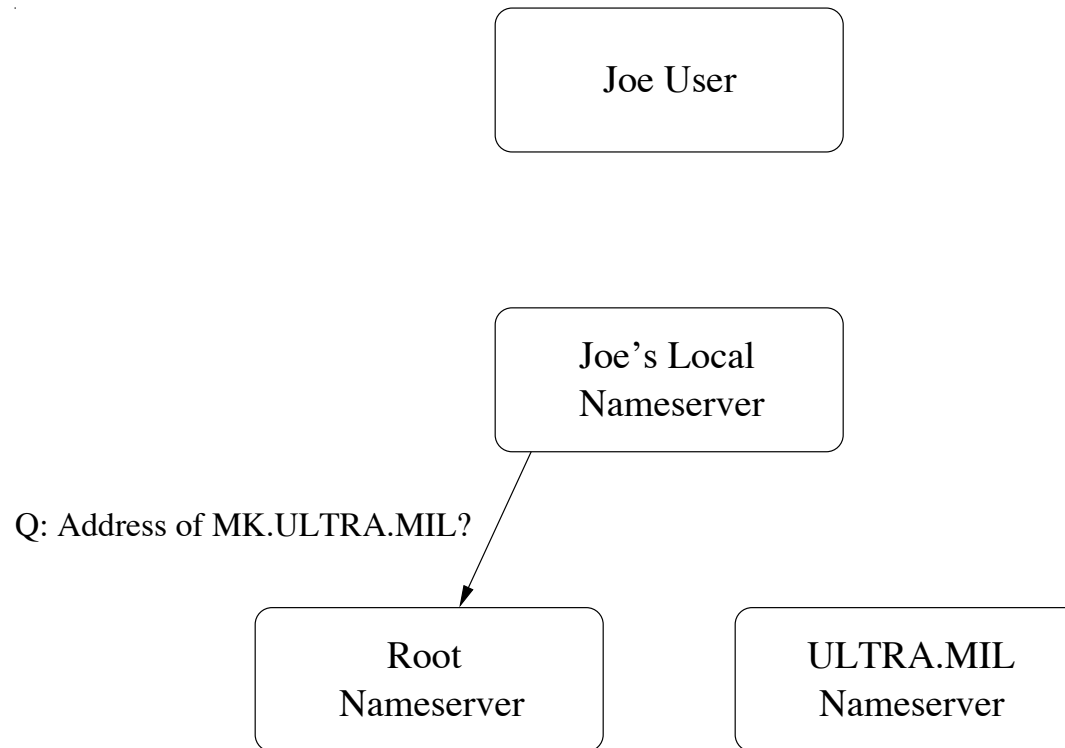
# Typical Forward Lookup

---



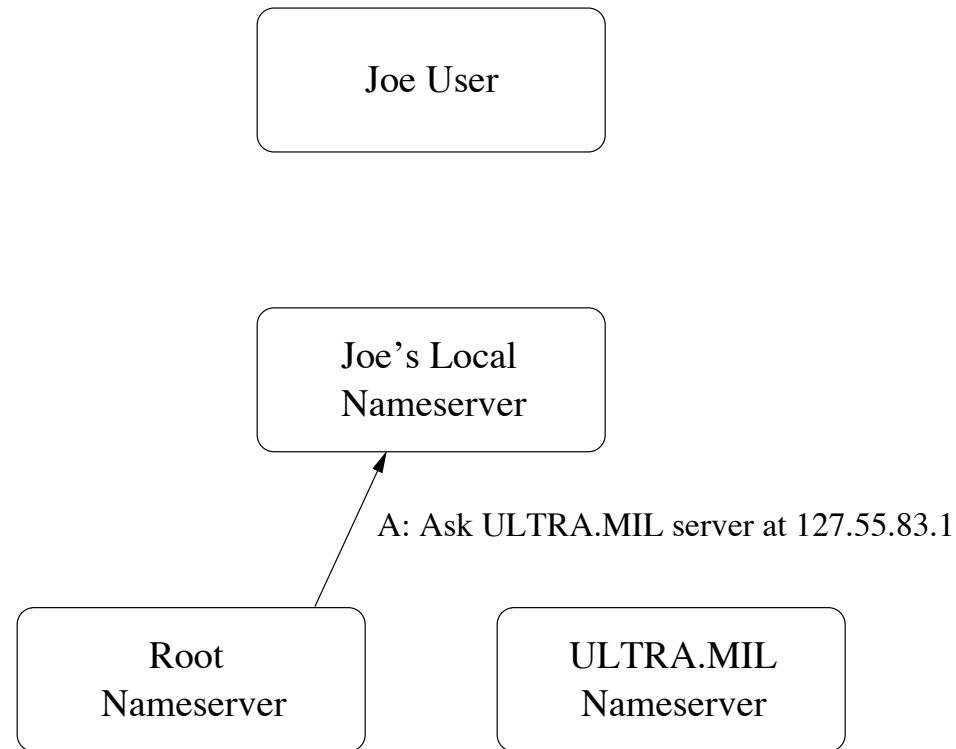
# Typical Forward Lookup

---



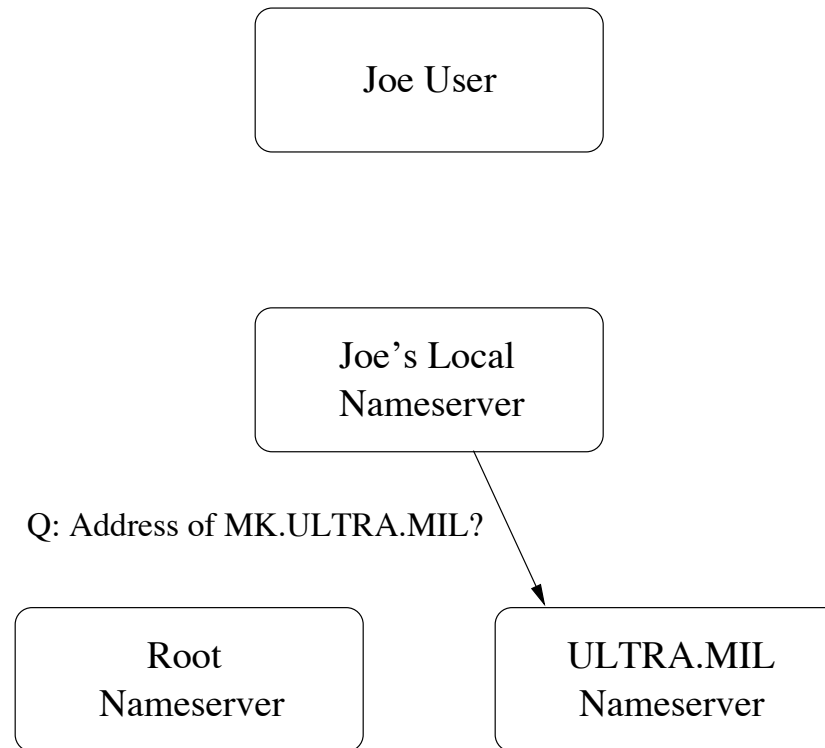
# Typical Forward Lookup

---



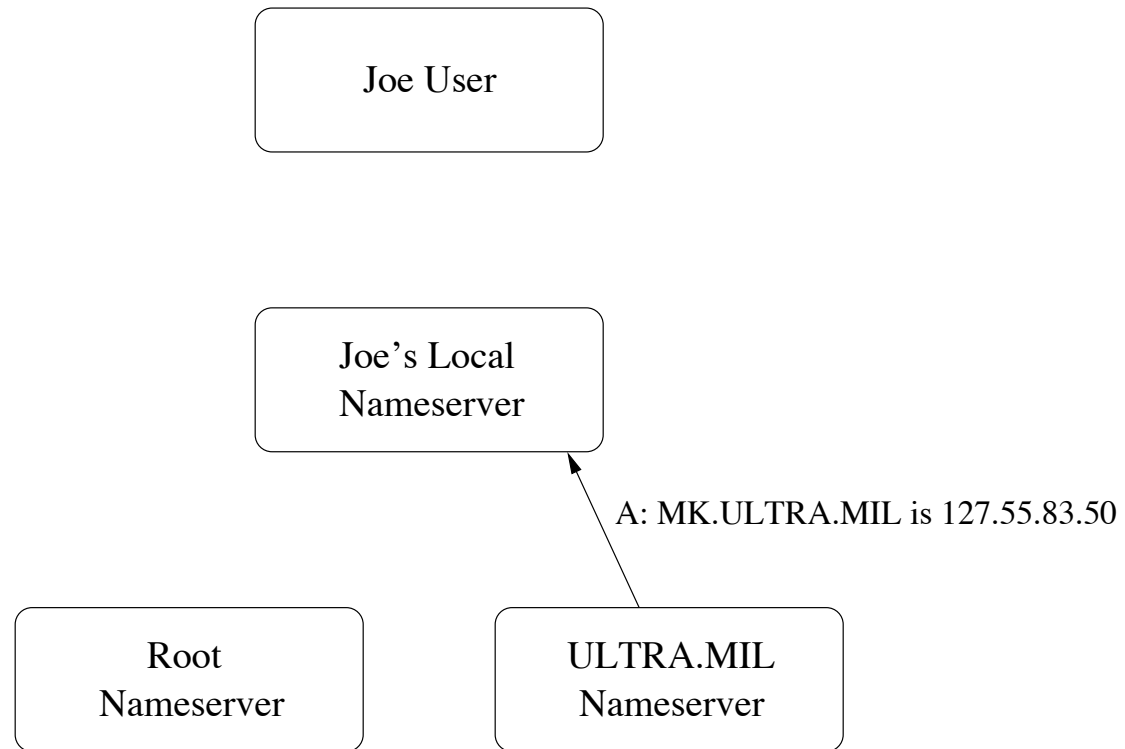
# Typical Forward Lookup

---



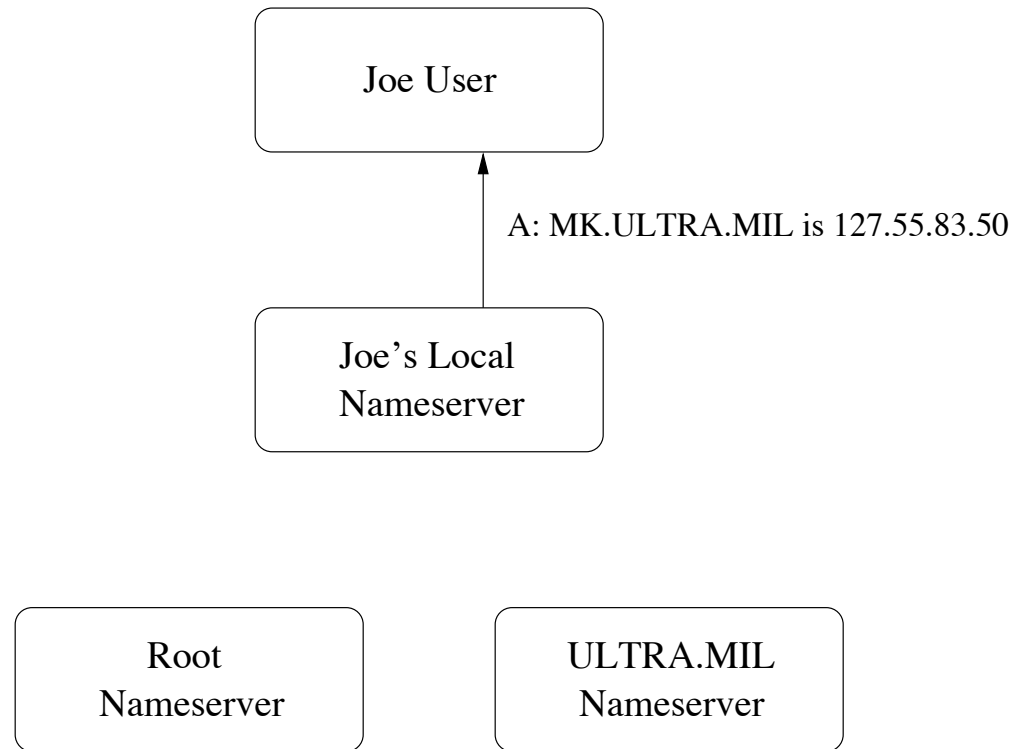
# Typical Forward Lookup

---



# Typical Forward Lookup

---



# Zone Files

---

- Nameservers store DNS information in a set of *zone files*, which contain *resource records*
- Resource records map a *name* to a *value*
  - *A* record: name to IPv4 address
  - *PTR* record: IP address to name
  - *CNAME* record: alias to canonical name
  - (and quite a few others...)







# IPv6 AAAA records

---

- Straightforward extension of A resource record from IPv4 to IPv6 addresses
- Defined in RFC 3596
- Can coexist with A records defined for the same hostname
- ...but this isn't the whole story



# IPv6 A6 Records

; Part of ULTRA.MIL. Zone file

mk	A6	64	::f23b:10ff:fe87:559d	ultra-net
ultra-net	A6	16	0:5b::	mil-net
mil-net	A6	0	2004::	

Hostname

Prefix Size

Prefix Name

Resource  
Record Type

IPv6 Address



pervasivet<sup>technology</sup>labs

AT INDIANA UNIVERSITY

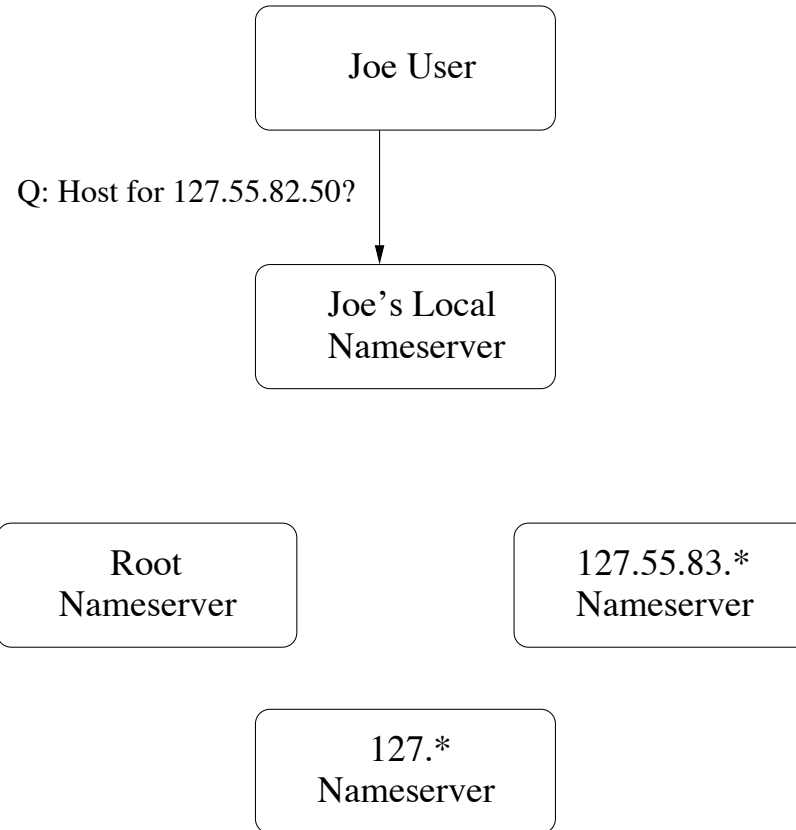
# IPv6 A6 Records

---

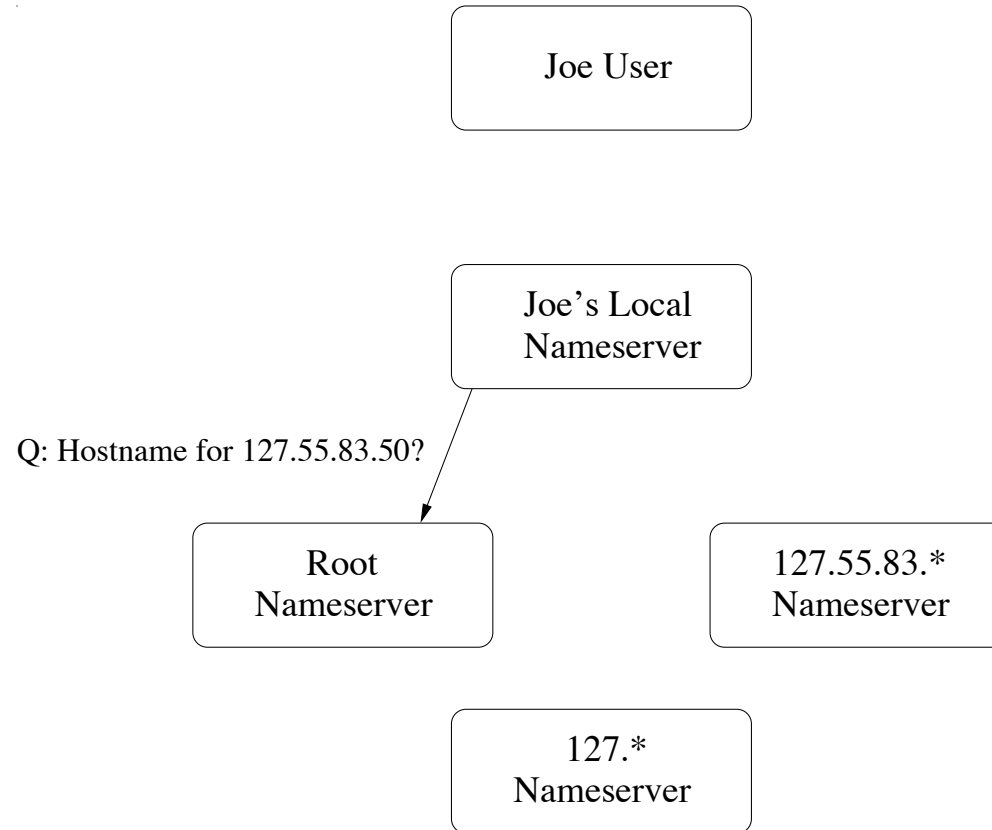
- Much more complicated than AAAA
- Defined in RFC 2874
- Prefix delegation can introduce external points of failure for every address lookup
- Additional danger of delegation loops
- Now relegated to experimental status and may disappear



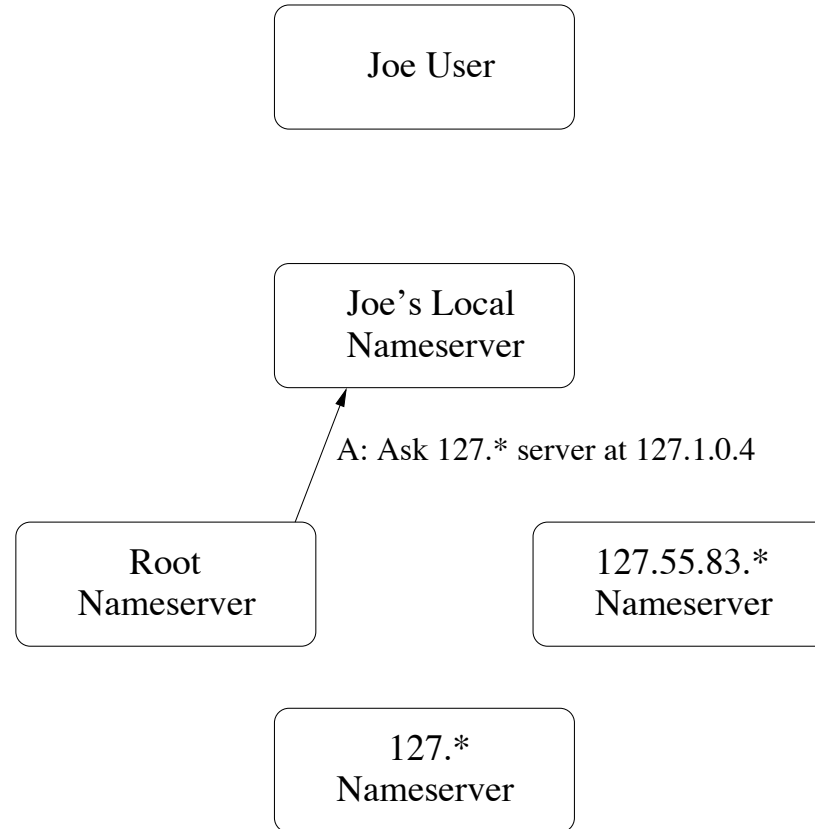
# Typical Reverse Lookup



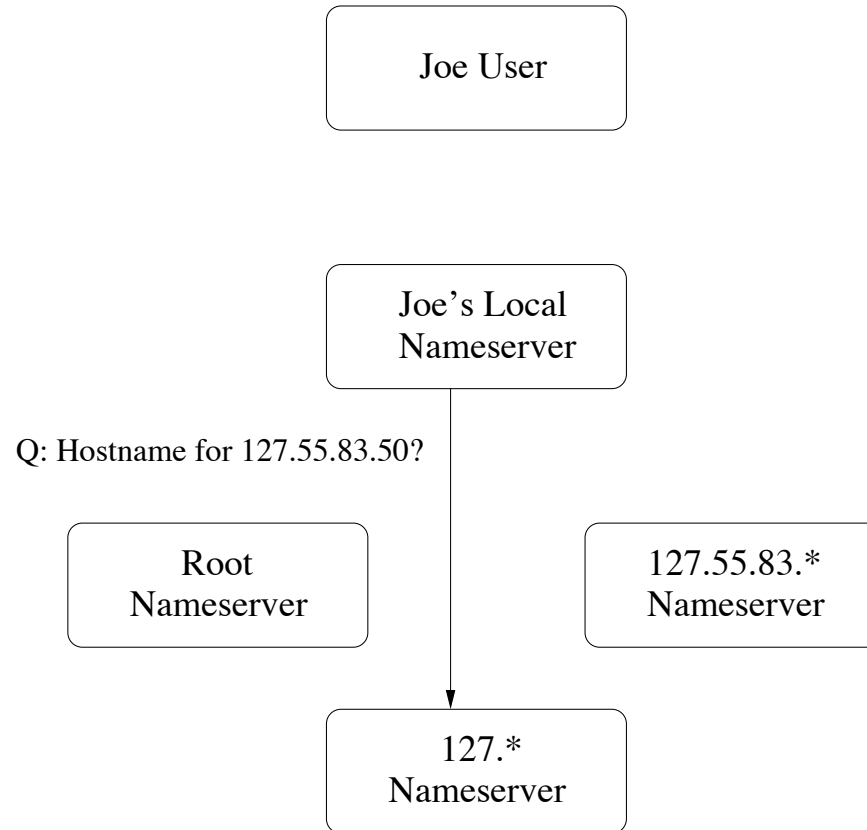
# Typical Reverse Lookup



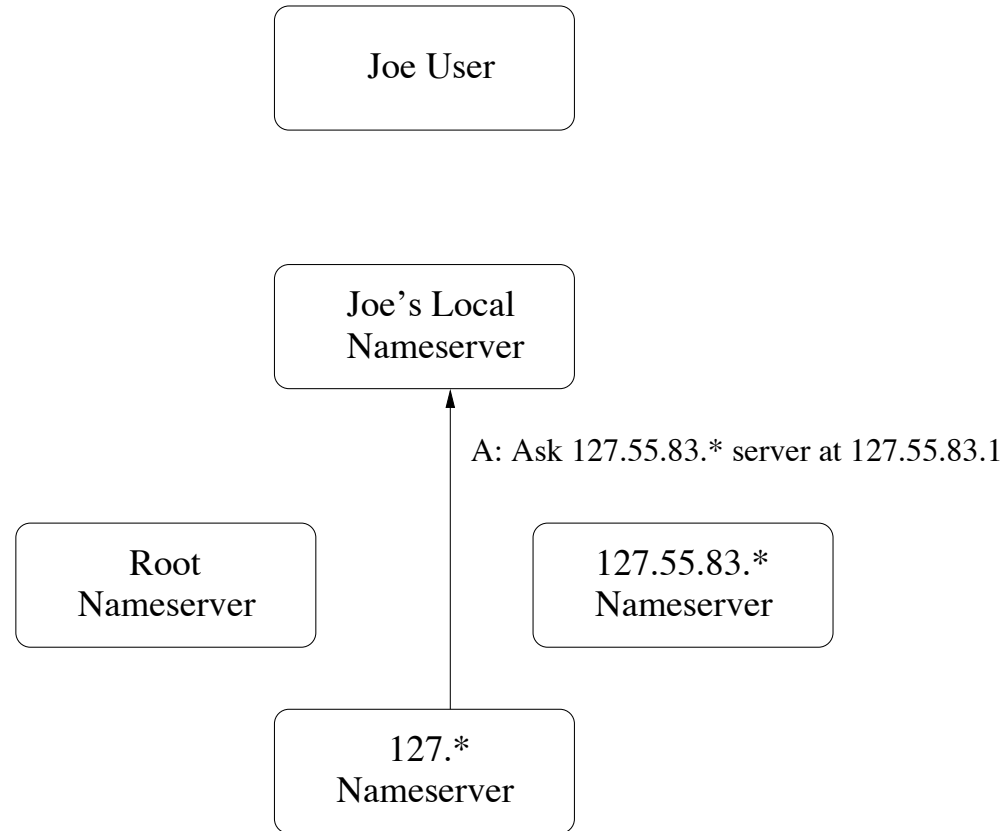
# Typical Reverse Lookup



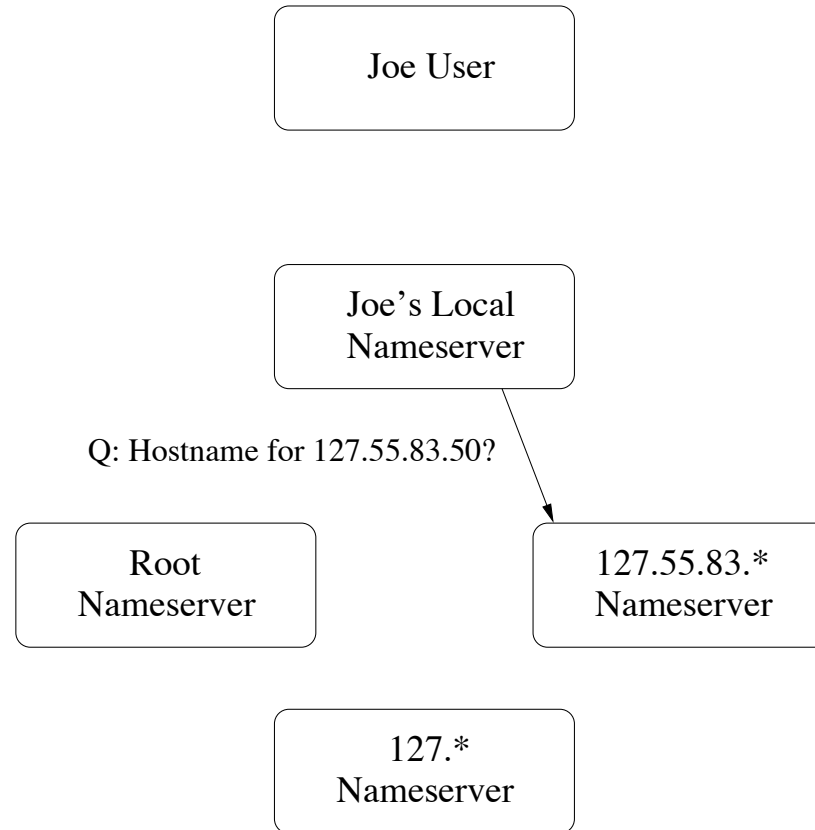
# Typical Reverse Lookup



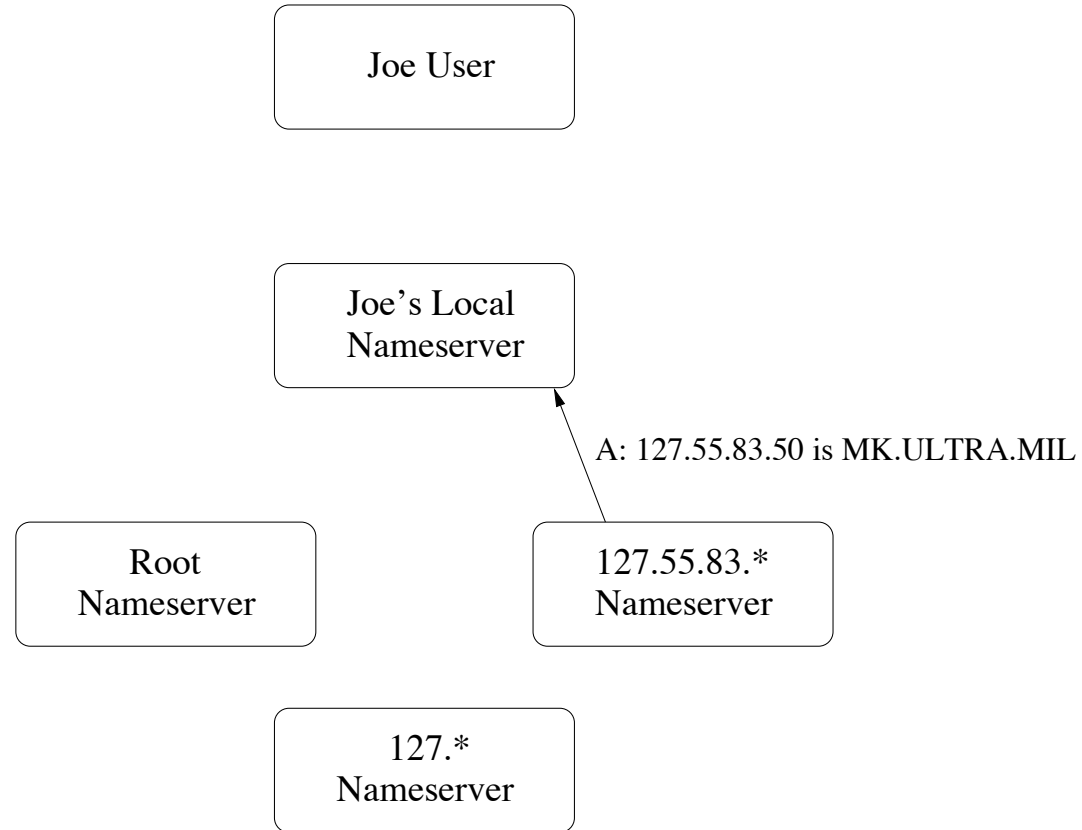
# Typical Reverse Lookup



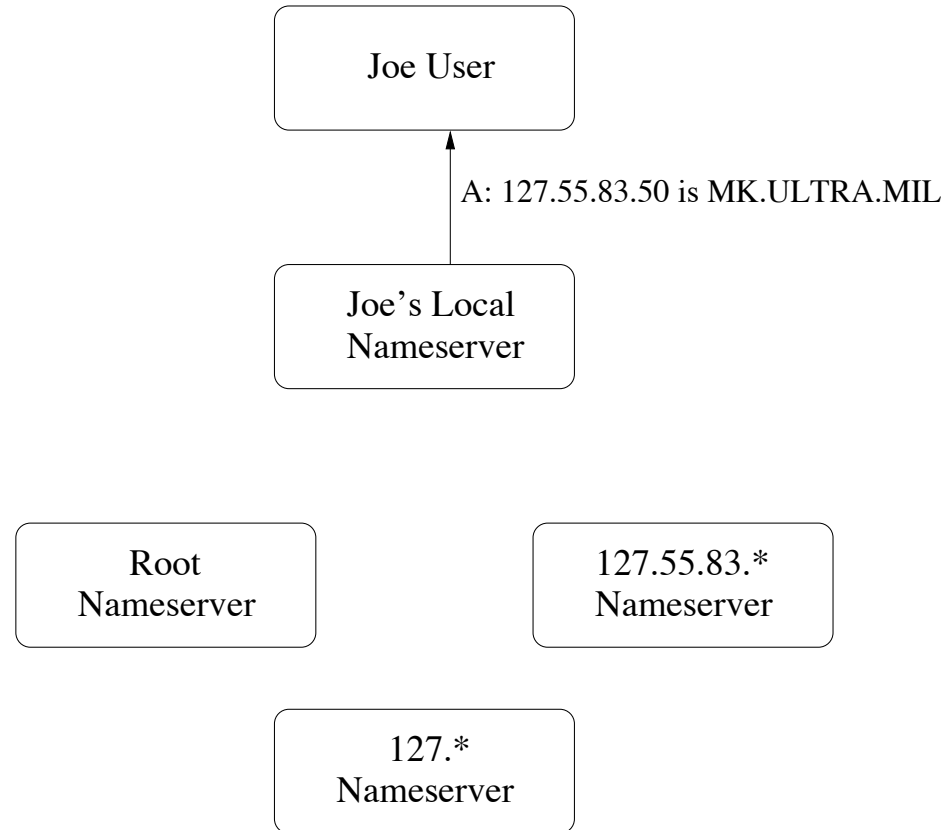
# Typical Reverse Lookup



# Typical Reverse Lookup



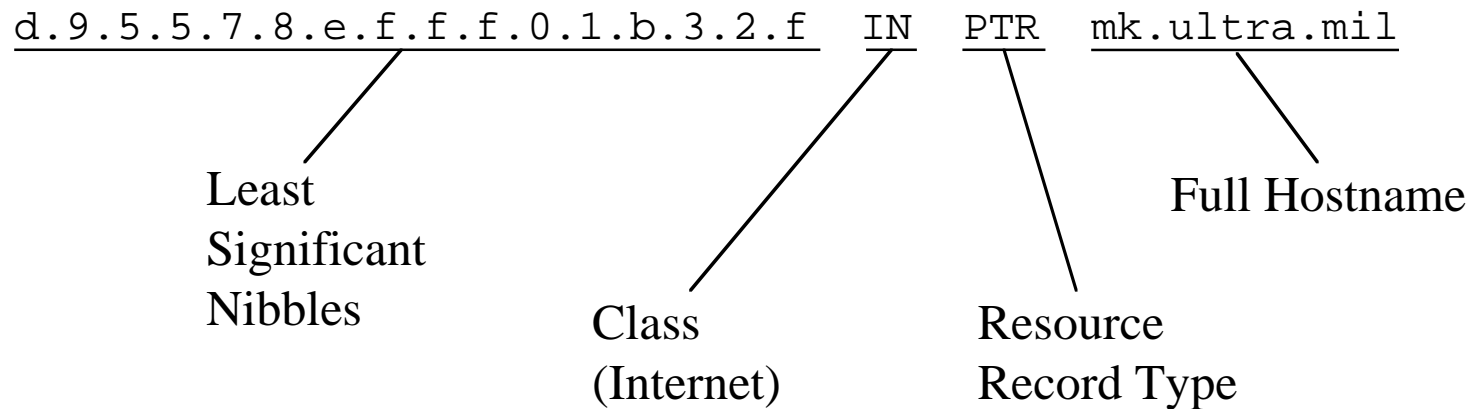
# Typical Reverse Lookup





# IPv6 PTR Records

; Part of 0.0.0.0.0.0.0.0.b.5.0.0.4.0.0.2.IP6.INT Zone file  
 ; OR 0.0.0.0.0.0.0.0.b.5.0.0.4.0.0.2.IP6.ARPA Zone file



# IPv6 PTR Records

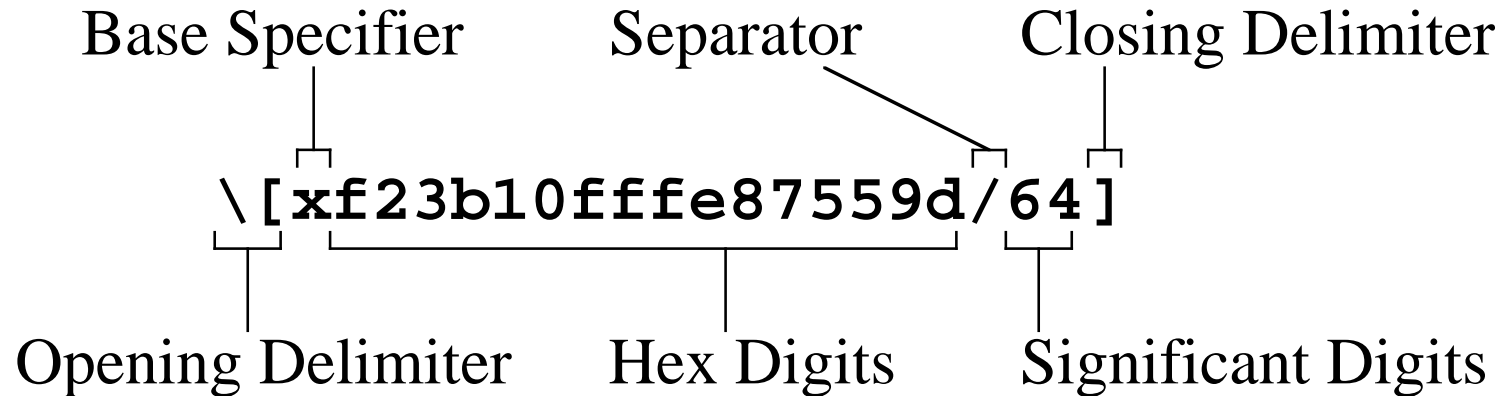
---

- Domain is either *IP6.INT* or *IP6.ARPA*
  - *IP6.ARPA* is superceding *IP6.INT* (RFC 3596)
- Address is split into 32 four-bit *nibbles*
  - Difficult to maintain in zone files
  - Potential increase in DNS queries by factor of 8
- Purpose of nibbles is to allow finer division of inverse domain
  - Still isn't enough granularity



# Bitstring Labels

---



# IPv6 *DNAME* Records

---

- Bitstring labels in *DNAME* records are used to give symbolic names to parts of the *PTR* record
  - These records can be stored on different nameservers
  - Subject to all the drawbacks of the *A6* record
  - Now relegated to experimental status as well

```
\[x2004/16].IP6.ARPA.          IN  DNAME  mil.  
\[x005b/16].mil.              IN  DNAME  ultra.mil.  
  
\[xf23b10fffe87559d/64].ultra.mil.  IN  PTR    mk.ultra.mil.
```



# Other DNS Concerns

---

- In general, you still need to do IPv6 queries using IPv4
  - There isn't a full set of IPv6 root nameservers
  - Some root servers (F-Root) do use native IPv6
- Still in rapid transition
  - AAAA vs. A6 debate has flipped several times
  - *.IP6.INT* vs. *.IP6.ARPA* is settling as well



# Other DNS Concerns

---

- Even after the transition period
  - Many IPv6 stacks will need to be updated
  - Possible need to support *all* of these options
- Greater latency possible for clients
  - Recursive queries can be much more complex
  - Potentially a greater load on nameservers
- Zone files become harder to maintain



# DNS Resolver Changes

---



pervasivet<sup>technology</sup>labs  
AT INDIANA UNIVERSITY

# What is the Resolver?

---

- The *resolver* is the standard programmer's API for working with DNS
- Built around two main calls:
  - `gethostbyaddr ( )`
  - `gethostbyname ( )`
- Other resolvers and APIs exist, but this is the one in the POSIX standard



# Standard Resolver Calls

---

```
struct hostent *gethostbyaddr  
    (const char *addr, int len, int type);
```

- *addr* is the address we're looking up
- *len* is size of that address
- *type* is the sort of address we're looking up (i.e., *AF\_INET*)

```
struct hostent *gethostbyname(const char *name);
```

- *name* is the hostname we're looking up



# Why Does It Need to Change?

---

- Inflexibility
  - `gethostbyname ( )` doesn't let you specify whether you want a IPv4 or IPv6 address
  - `gethostbyaddr ( )` doesn't let you specify whether you want an AAAA or A6 record
  - `struct hostent` holds an array of addresses, not a linked list
    - The address type must be homogeneous



# Why Does It Need to Change?

---

- Reentrancy
  - The current calls are not thread-safe
  - The pointers returned are to static data maintained by the resolver itself and reused for each call
  - This means than an application using this API *must* serialize all DNS lookups
    - Ever wonder why IE and Netscape hang for a while?



# Quick Fix

---

```
gethostbyname2(const char *name, int af);
```

- *name* is the hostname we're looking up
- *af* is the address family for the lookup (*AF\_INET*, etc.)
  
- This call is a GNU extension
- This does not resolve any reentrancy issues
- However, it's quick and easy



# Another Fix

---

```
int gethostbyname_r  
    (const char *name, struct hostent *ret,  
     char *buf, size_t buflen,  
     struct hostent **result, int *h_errnop);
```

- *name* is the hostname we're looking up
- *ret* stores the result of the call
- *buf* stores chunks of auxiliary data
- *buflen* specifies the size of *buf*
- *result* uses bit of *buf* to store host structures
- *h\_errnop* holds any error number generated



# Another Fix

---

- The technical expression for this interface is “horrid train wreck”
- This is also a GNU extension
- There is also a `gethostbyname2_r ( )` call that works similarly
  - Adds the ability to specify address family



# POSIX Fix

---

- POSIX 1003.1-2001 classified `gethostbyaddr ( )` and `gethostbyname ( )` as legacy calls
  - Replacements are `getipnodebyaddr ( )` and `getipnodebyname ( )`
- The interface to this functions is irrelevant, because you probably don't have them



# The Real Fix

---

```
int getaddrinfo
    (const char *node, const char *service,
     const struct addrinfo *hints,
     struct addrinfo **res);
```

- *node* is the hostname or address we're looking up
- *service* is used to propagate a port number into returned structures
- *hints* is used to select address family and other lookup options
- *res* is a pointer to the pointer that holds the address of the results



# Code Example

---

```
struct addrinfo  hints, *info;
int              status;
const char       *host = "mk.ultra.mil";
const char       *port = "7734";

// set up the hints
memset(&hints, 0, sizeof(hints));
hints.ai_family   = AF_INET6;
hints.ai_socktype = SOCK_STREAM;
```



# Code Example

---

```
// do the lookup
status = getaddrinfo(host, port, &hints, &info);
if (status)
    exit(EXIT_FAILURE);

// . . . do something with the information . . .

// free the memory
freeaddrinfo(info);
```



# Why to Use `getaddrinfo()`

---

- Handles multiple address families correctly
  - Includes ability to retrieve both IPv4 and IPv6 addresses simultaneously
- Thread-safe
- Straightforward memory management
- Already fairly portable
- Initializes socket address structures directly



# Conclusion

---

- IPv6 is a *lot* more than just bigger addresses
  - Everyone jumped in and tried to add the features they've always thought would be nifty
  - The results are complex, conflicting standards and inconsistent feature sets
  - This is especially true of DNS
- More changes are likely to come

