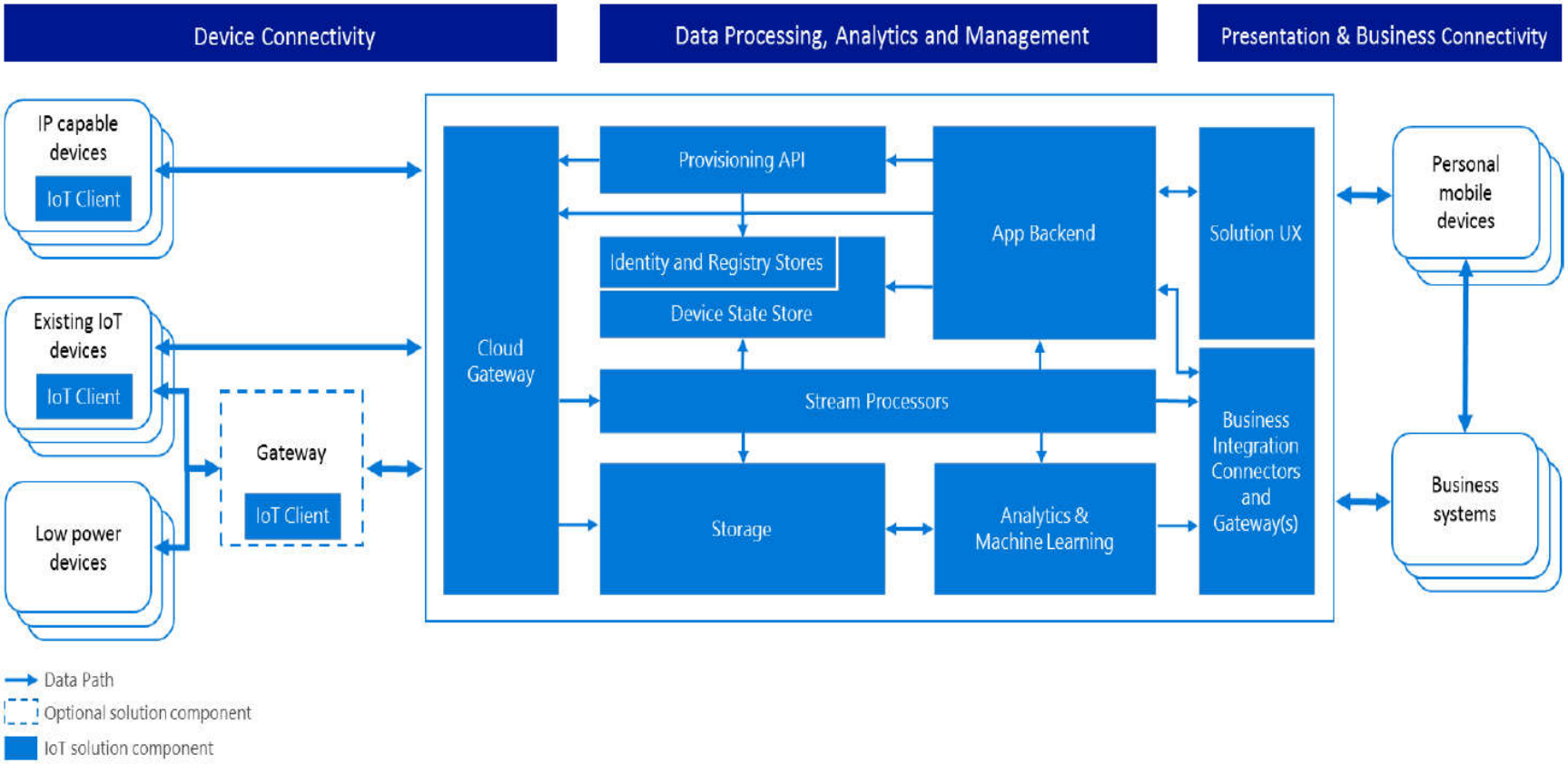
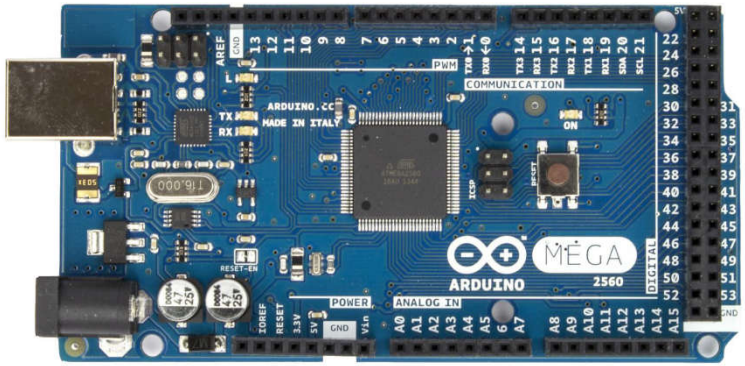
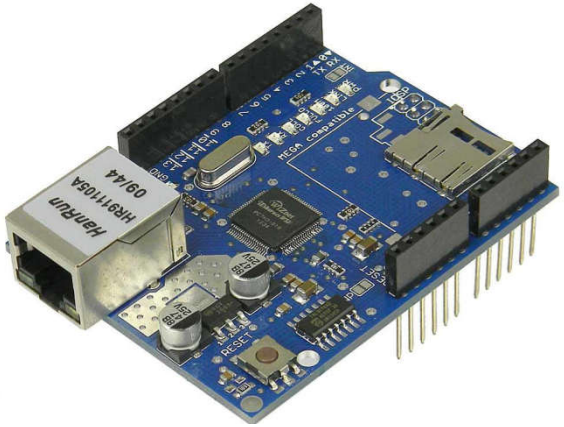
Multithreading

Cloud centric concept

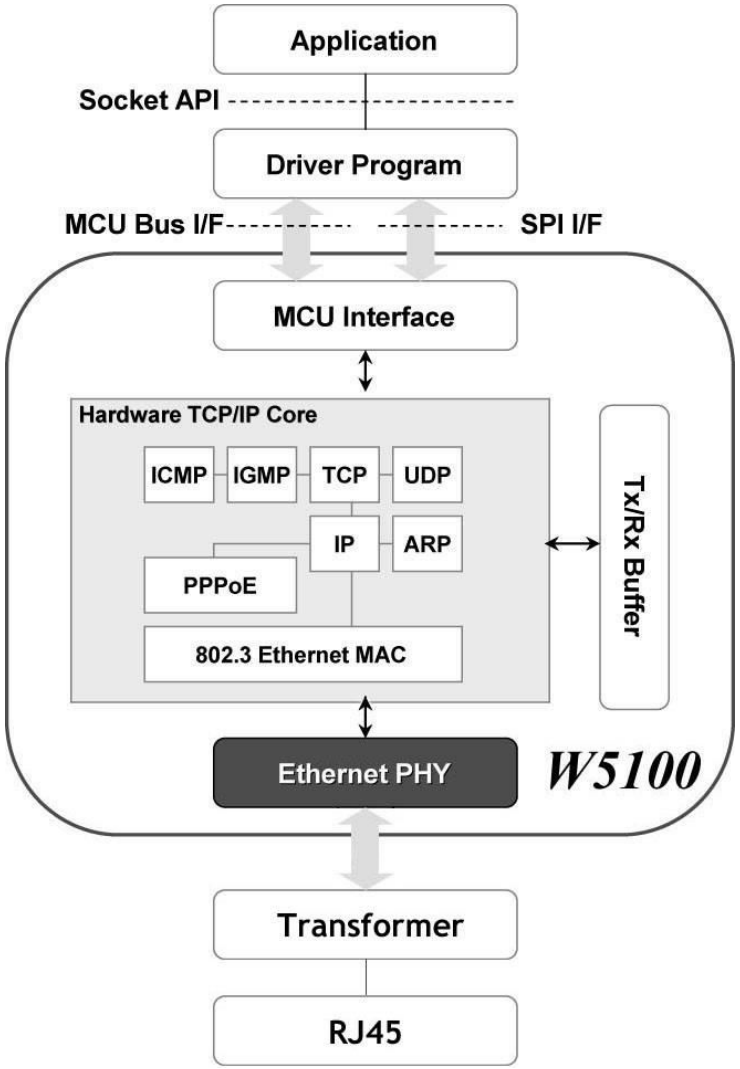


Cloud centric concept





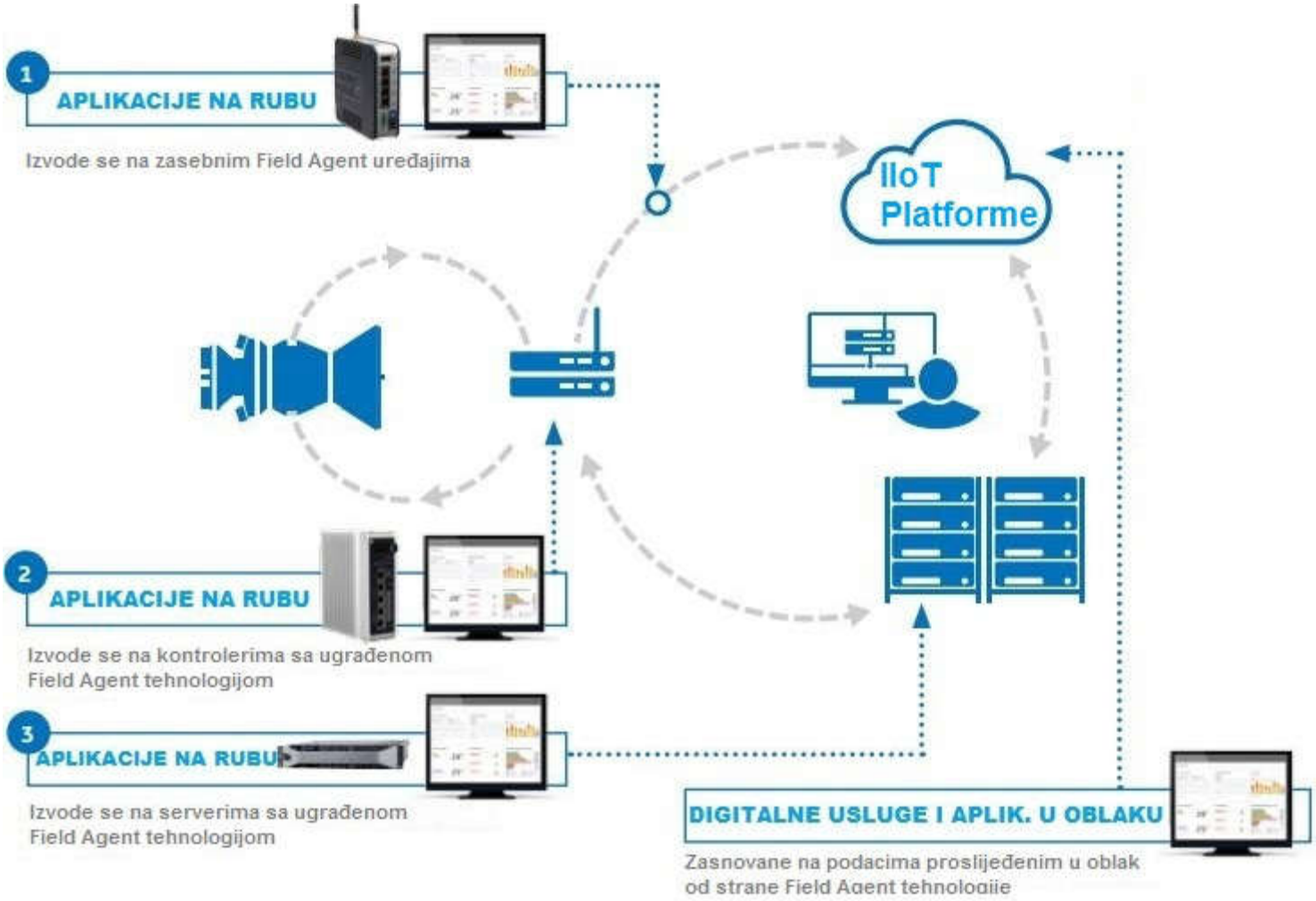
Arduino Ethernet Shield



Fog computing: Field agent

•Field Agent is the critical link required in an IIoT chain for cloud-enabled analytics.
•provides a rugged, pre-configured solution for secure data collection and conveyance from the machine.
•Connect to any industrial asset in order to collect data, analyze trends and uncover insights that improve operations and asset performance.
•To build out remote monitoring & diagnostics capabilities safely and securely, utilizing encrypted channels that preserve data time stamp, quality and fidelity.

Fog computing: Field agent



Small IoT network

Single server

Multitasking

•In computing, multitasking is a method by which multiple tasks, also known as processes, share common processing resources such as a CPU.
•With a multitasking OS you can “simultaneously” run multiple applications.
•Multitasking refers to the ability of the OS to quickly switch between each computing task to give the impression the different applications are executing multiple actions simultaneously.

Multithreading

• Multithreading extends the idea of multitasking into applications, so you can subdivide specific operations within a single application into individual threads.
• Each of the threads can run in parallel.
•The OS divides processing time not only among different applications, but also among each thread within an application.

Applications that take advantage of multithreading have numerous benefits, including the following:
• More efficient CPU use
• Better system reliability
• Improved performance on multiprocessor computers

Nonblocking I/O

• The default behavior of a socket call is to block until the requested action is completed.
• process with a blocked function is suspended by the operating system.
• solution to the problem of undesirable blocking is to change the behavior of the socket so that all calls are *nonblocking*
• if a requested operation can be completed immediately, the call's return value indicates success; otherwise it indicates failure (usually -1).
• change of the default blocking behavior:

int fcntl(**int** *socket,* **int** *command,* **long** *argument)*

Commands: F_GETFL and F_SETFL
Argument O_NONBLOCK.

Advantages of Threads

- The overhead for creating a thread is significantly less than that for creating a process
- Multitasking, i.e., one process serves multiple clients
- Switching between threads requires the OS to do much less work than switching between processes

Drawbacks of Threads

- Not as widely available as longer established features
- Writing multithreaded programs require more careful thought
- More difficult to debug than single threaded programs
- For single processor machines, creating several threads in a program may not necessarily produce an increase in performance

Thread Synchronization Mechanisms

- Mutual exclusion (mutex):
  - guard against multiple threads modifying the same shared data simultaneously
  - provides locking/unlocking critical code sections where shared data is modified
  - each thread waits for the mutex to be unlocked (by the thread who locked it) before performing the code section

Basic Mutex Functions

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
    *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- a new data type named pthread_mutex_t is designated for mutexes
- a mutex is like a key (to access the code section) that is handed to only one thread at a time
- the attribute of a mutex can be controlled by using the pthread_mutex_init() function
- the lock/unlock functions work in tandem

```
#include <pthread.h>
...
pthread_mutex_t my_mutex;
...
int main()
{
        int tmp;
        ...
        // initialize the mutex
        tmp = pthread_mutex_init( &my_mutex, NULL );
        ...
        // create threads
        ...
        pthread_mutex_lock( &my_mutex );
            do_something_private();
        pthread_mutex_unlock( &my_mutex );
        ...
        …
        pthread_mutex_destroy(&my_mutex );
        return 0;
}
```

- Whenever a thread reaches the lock/unlock block, it first determines if the mutex is locked. If so, it waits until it is unlocked. Otherwise, it takes the mutex, locks the succeeding code, then frees the mutex and unlocks the code when it's done.

Semaphores

- Counting Semaphores:
  - permit a limited number of threads to execute a section of the code
  - similar to mutexes
  - should include the semaphore.h header file
  - semaphore functions do not have pthread_ prefixes; instead, they have sem_ prefixes

Basic Semaphore Functions

- creating a semaphore:

int sem_init(sem_t *sem, int pshared, unsigned int value);

  - initializes a semaphore object pointed to by sem
  - pshared is a sharing option; a value of *0* means the semaphore is local to the calling process
  - gives an initial value value to the semaphore

- terminating a semaphore:

int sem_destroy(sem_t *sem);

  - frees the resources allocated to the semaphore sem
  - usually called after pthread_join()
  - an error will occur if a semaphore is destroyed for which a thread is waiting

Basic Semaphore Functions

- semaphore control:

```
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
```

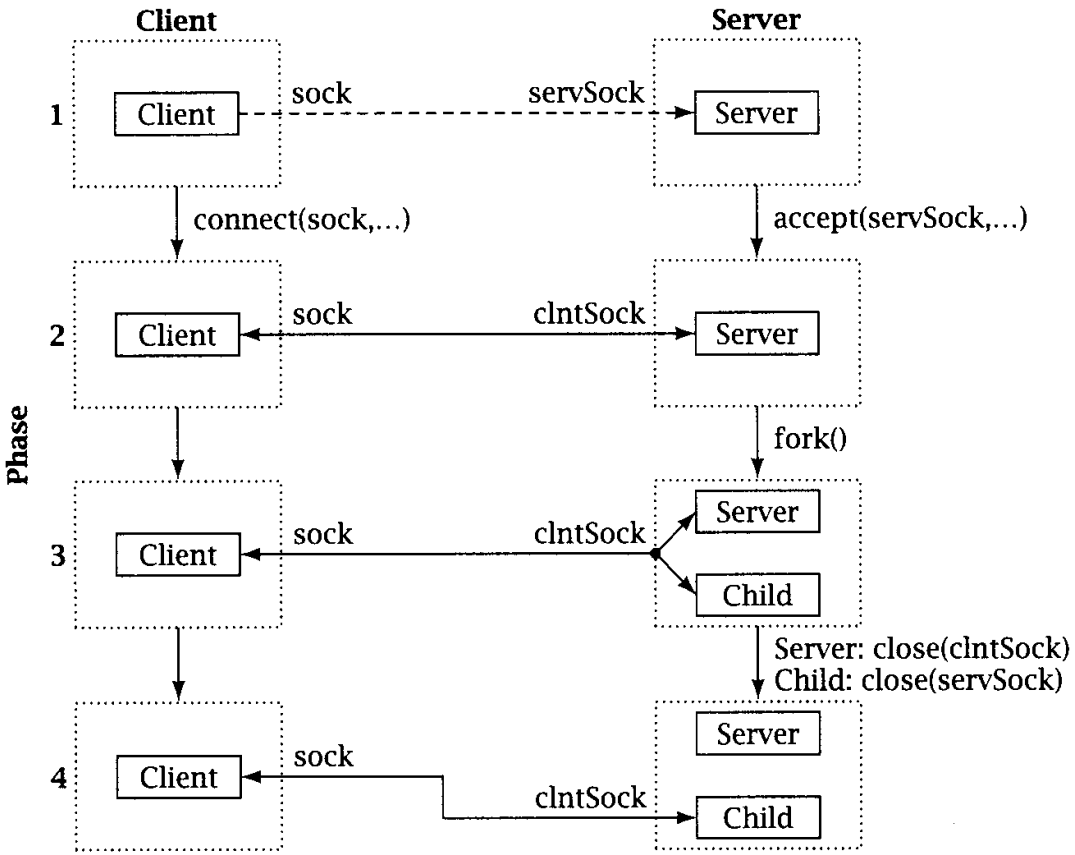
  - sem_post *atomically* increases the value of a semaphore by 1, i.e., when 2 threads call sem_post simultaneously, the semaphore's value will also be increased by 2 (there are 2 atoms calling)
  - sem_wait *atomically* decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first

```
#include <pthread.h>
#include <semaphore.h>
...
void *thread_function( void *arg );
...
sem_t semaphore;        // also a global variable just like mutexes
...
int main()
{
        int tmp;
        ...
        // initialize the semaphore
        tmp = sem_init( &semaphore, 0, 0 );
        ...
        // create threads
        pthread_create( &thread[i], NULL, thread_function, NULL );
        ...
        while ( still_has_something_to_do() )
        {
            sem_post( &semaphore );
            ...
        }
        ...
        pthread_join( thread[i], NULL );
        sem_destroy( &semaphore );
        return 0;
}
```

```
void *thread_function( void *arg )
{
        sem_wait( &semaphore );
        perform_task_when_sem_open();
        ...
        pthread_exit( NULL );
}
```

- the main thread increments the semaphore's count value in the while loop
- the threads wait until the semaphore's count value is non-zero before performing perform_task_when_sem_open() and further
- daughter thread activities stop only when pthread_join() is called

Asynchronous I/O
Per-Client Process



Asynchronous I/O
Per-Client Thread