Exam Questions

Type 1
Consider a 1000m² area with a uniform distribution of 1000 sensor nodes, each equipped with a temperature sensor. You want to constitute a temperature map of the area at various resolutions. The minimum resolution is the average temperature for the entire network, the maximum resolution is the temperature for each sensor. Describe a solution to this task.

Type 1
Consider the following sensor network deployed in a greenhouse: a backbone of sensor nodes are powered on the electrical grid, each backbone node is connected with up to 3 other backbone nodes and up to 5 sensor nodes. The sensor nodes are battery powered and equipped with humidity sensors placed in the ground. You want to alert the farmer in case of abnormal humidity. Lifetime is a crucial property for this sensor network. Describe a solution to this task.

Type 2
Consider the marketing document promoting Embernet/Zigbee in Appendix A. What are the pros and cons of Embernet/Zigbee in the context of the sensor network regime.

(A way to answer this question is to define the sensor network regime and analyze the information this document contains (or does not contain) based on this definition)

Type 2
Consider the marketing document promoting QNX RTOS in Appendix B. What are the pros and cons of QNX RTOS in the context of the sensor network regime.

Type 2
Consider the short description of the Java KVM in Appendix C. What are the pros and cons of QNX RTOS in the context of the sensor network regime.

Type 3
Give an example network topology where GPSR cannot find a way from a source to a sink. Explain your answer.

Type 3
What problems do you foresee when scaling TinyDB to a 1000 node sensor network?
Appendix A – Embernet/Zigbee

Until now, using wireless networking solutions for sensing and control applications, although desirable, was impractical. Wireless networks were problematic in many challenging environments because of the limitations of the traditional wireless network architectures, which left networks vulnerable to physical obstructions, radio interference, single points of failure and bottlenecks.

EmberNet networks use flexible, multi-hop networking that can follow several architectural topologies, to ensure that your network functions with maximum efficiency and reliability.

For maximum flexibility and reliability, EmberNet’s mesh topology, where each node is in direct communication with its immediate neighbors, is the right choice. If a single node fails for any reason—including the introduction of strong RF interference—messages are automatically routed through alternate paths.

Networks based on a star topology will provide efficient localized (one-hop) communication. In these networks, a central access point controls communication between nodes. However, when physical or RF interference blocks communication between the access point and any of the nodes, the network cannot recover until the source of the interference is removed.

A superstar configuration can combine the benefits of both mesh and star topologies. Perfect for two-tier networks, where local star nodes are controlled by mesh nodes, the superstar topology provides both efficiency and flexibility.

The topologies supported by EmberNet provide a wireless network with the right mix of flexibility, reliability, and efficiency required by your sensing and control applications.
Appendix B – QNX Real Time Operating System (RTOS)

**Microkernel Architecture**
With its true microkernel architecture, the QNX® Neutrino® RTOS can run every driver, application, protocol stack, and file system in the safety of memory-protected user space. Moreover, it uses message passing as the fundamental means of IPC, allowing your system to achieve exceptional scalability, modularity, and ease of design.

**POSIX Support**
The QNX Neutrino RTOS has been engineered from the ground up for POSIX standards, allowing you to quickly port Linux, Unix, and other open source programs. QNX Neutrino complies with POSIX 1003.1-2001 and offers an extensive POSIX feature set, including realtime extensions and threads.

**Power Management Framework**
Using this comprehensive framework, you can exercise fine-grained control over the power states of every peripheral and create a customized, application-specific power policy for each system you build. The framework includes libraries to build power-managed drivers, power-sensitive applications, and a centralized power manager.

**Symmetric Multiprocessing Microkernel**
QNX Neutrino is the only RTOS to support symmetric true multiprocessing, which offers enormous scaling benefits for network elements, image processing applications, and other compute-intensive systems. Better yet, QNX Neutrino lets you take advantage of SMP without having to hardcode SMP awareness into your software.

**Instrumented Kernel**
Using the instrumented kernel, you can quickly pinpoint timing conflicts, deadlocks, logic flaws, software faults, and a variety of other hotspots, in both uniprocessor and multiprocessor systems.

**Critical Process Monitor**
A “smart watchdog” that extends QNX Neutrino’s inherent fault isolation, the critical process monitor lets your system recover from faults automatically and achieve faster Mean Time to Repair. Supports instant fault notification, customized failure recovery, instant reconnections, and postmortem analysis.

**Graphical User Interface**
Highly modular, the QNX Photon microGUI® windowing system can bring sophisticated graphics to small-footprint devices, including support for advanced multilayer displays.

- customizable look-and-feel
- support for multilingual interfaces
- scalable fonts for low-res displays
- off-the-shelf and customized media formats
- embedded web browser support

**Transparent Distributed Processing**
Transparent distributed processing provides a framework for unified access to hardware and
software resources on remote nodes, using standard messages. The net effect: any remote resource — disks, NICs, protocol stacks, and so on — can all be accessed as if they were on the local machine.

- **IP Networking Technologies**
  Networking services execute outside the kernel as separate, memory-protected processes, so you can start, stop and upgrade any network driver or protocol on the fly. You can also choose from a range of protocol stacks, including NetBSD (IPsec, IPv6) and tiny TCP/IP stacks, and run multiple instances of a stack on the same physical interface.

- **Java Environments**
  QNX supports IBM Websphere Custom Environment (runtimes) for embedded Java development. In the QNX Neutrino RTOS, Java applications get full access to distributed processing, symmetric multiprocessing, the QNX Photon microGUI, and other advanced OS services.

- **File Systems**
  File systems execute outside the kernel, in memory-protected user space. As a result, you can start, stop or upgrade file systems on the fly, without having to reboot. QNX Neutrino supports a complete range of flash, embedded, disk, network, and special file systems.

- **Resource Manager Framework**
  If you’re writing a driver for custom hardware or developing any service that communicates with multiple client applications, the resource manager framework will simplify your task. The framework provides a library that eases the setup and teardown of communication channels, along with a thread-pool library that makes multithreaded drivers smaller and easier to write.

- **Driver Development Kits**
  The QNX Momentics® development suite Professional Edition offers a variety of driver development kits (DDKs) to create custom drivers for audio, character, graphics, input, printer, networking, parallel, serial, and USB devices. Because QNX drivers run as standard, memory-protected processes, you can debug and optimize them with source-level tools, and test them without having to repeatedly rebuild and reboot.

- **Processor Support**
  The QNX Neutrino RTOS offers advanced support for the ARM, MIPS, PowerPC, SH-4, StrongARM, Intel® XScale™ Microarchitecture, and x86 processor families. It also supports functions and macros that let you write processor-independent drivers and applications. You can debug and test your applications before the target processor is chosen or target different processors.
A Java virtual machine is the foundation for Java technology, allowing applications written in the Java programming language to be portable across different hardware environments and operating systems. The virtual machine mediates between the application and the underlying platform, converting the application's bytecodes into machine-level code appropriate for the hardware and operating system being used. In addition to governing the execution of an application's bytecodes, the virtual machine handles related tasks such as managing the system's memory, providing security against malicious code, and managing multiple threads of program execution.

In order to meet the market need for a very small footprint Java implementation, the KVM was designed to overcome three key technical challenges: reducing the size of the virtual machine and class libraries themselves, reducing the memory utilized by the virtual machine during execution, and allowing for components of the virtual machine to be configured to suit particular devices (for example, by allowing pluggable garbage collection).

The KVM design team employed a number of strategies in overcoming these technical challenges. One of these strategies was to enable the partitioning of virtual machine capabilities. The team also architected the byte code interpreter and garbage collector to minimize dynamic memory usage, and carefully implemented the virtual machine and libraries to minimize their size (e.g. by rolling Java Native Interface calls into the VM itself).

The results of the careful design and implementation are readily apparent:

- **Reduced VM Size** - the K virtual machine is currently only 50-80 K of object code in its standard configuration, depending on target platform and compilation options.
- **Reduced Memory utilization** - In addition to the K virtual machine's small object size, it requires only a few tens of kilobytes of dynamic memory to run effectively. Because of the reduced VM size and memory utilization, even with total memory available of only 128K the K virtual machine enables useful Java technology-based applications to run on a device.
- **Performance** - the K virtual machine is able to run effectively on 16 bit processors clocked as low as 25 MHz, and can scale smoothly up to much more powerful 32 bit processors.
- **Portability** - although implemented in native code for extra performance, the K virtual machine has a highly portable architecture that reduces system dependencies to a minimum. Even multi-threading and garbage collection have been implemented in a completely system-independent manner, enabling speedy porting to any host platform.